Post-Quantum Verification of Fujisaki-Okamoto

Dominique Unruh
University of Tartu

August 6, 2020

Abstract

We present a computer-verified formalization of the post-quantum security proof of the Fujisaki-Okamoto transform (as analyzed by Hövelmanns, Kiltz, Schäge, and Unruh, PKC 2020). The formalization is done in quantum relational Hoare logic and checked in the `qrhl-tool` (Unruh, POPL 2019).

Contents

1 Introduction

In this paper, we present the first formal verification of the post-quantum security of the Fujisaki-Okamoto transform.

Cryptographic security proofs tend to be complex, and, due to their complexity, error prone. Small mistakes in a proof can be difficult to notice and may invalidate the whole proof. For example, the proof of the OAEP construction [8] went through a number of fixes [19, 20, 34] until it was finally formally proven in [5] after years of industrial use. The PRF/PRP switching lemma was a standard textbook example for many years before it was shown that the standard

proof is flawed [10]. And more recently, an attack on the ISO standardized blockcipher mode OCB2 [25] was found [24], even though OCB2 was believed to be proven secure by [31].

While a rigorous and well-structured proof style (e.g., using sequences of games as advocated in [10, 35]) can reduce the potential for hidden errors and imprecisions, it is still very hard to write a proof that is 100% correct. (Especially when proof techniques such as random oracles [13] or rewinding [42, 45] are used.) And especially if a mistake in a proof happens in a step that seems very intuitive, it is quite likely that the mistake will also not be spotted by a reader.

This problem is exacerbated in the case of post-quantum security (i.e., security against quantum adversaries): Post-quantum security proofs need to reason about quantum algorithms (the adversary). Our intuition is shaped by the experience with the classical world, and it is easy to have a wrong intuition about quantum phenomena. This makes it particularly easy for seemingly reasonable but incorrect proof steps to stay undetected in a post-quantum security proof.

In a nutshell, to ensure high confidence in a post-quantum security proof, it is not sufficient to merely have it checked by a human. Instead, we advocate formal (or computer-aided) verification: the security proof is verified by software that checks every proof step. In this paper, we present the first such formal verification, namely of a variant of the Fujisaki-Okamoto transform [18] as analyzed by Hövelmanns, Kiltz, Schäge, and Unruh [23].

Post-quantum security. Quantum computers have long been known to be a potential threat to cryptographic protocols, in particular public key encryption. Shor's algorithm [33] allows us to efficiently solve the integer factorization and discrete logarithm problems, thus breaking RSA and ElGamal and variants thereof. This breaks all commonly used public key encryption and signature schemes. Of course, as of today, there are no quantum computers that even come close to being able to execute Shor's algorithm on reasonable problem sizes. Yet, there is constant progress towards larger and more powerful quantum computers (see, e.g., the recent breakthrough by Google [3]). In light of this, it is likely that quantum computers will be able to break today's public key encryption and signature schemes (and possibly other kinds of cryptosystems) in the foreseeable future. Since the development, standardization, and industrial deployment of a cryptosystem can take many years, we need to develop and analyze future post-quantum secure protocols already today. One important step in this direction is the NIST post-quantum competition [29] that will select a few post-quantum public-key encryption and signature schemes for industrial standardization.

Quantum random oracles. One important proof technique in cryptography are random oracles [9]. In a proof in the random oracle model, we idealize hash functions by assuming that every hash function is simply a uniformly random function. (All algorithms including the adversary get oracle access to that function.) Based on this assumption, security proofs become considerably simpler. In some cases, we only know security proofs in the random oracle model. Of course, this comes at a cost: This assumption is an idealization; concluding that a protocol that is secure in the random oracle model is also secure using a real-world hash function is merely a heuristic argument. (And this heuristic is known to be false in certain contrived cases, e.g., [15].)

As first explicitly pointed out by [13], in the quantum setting, the random oracle becomes more involved: To get a realistic modeling, the adversary needs to be given superposition access to the random oracle, i.e., the adversary can evaluate the random oracle/hash function in a quantum superposition of many possible inputs. Due to this, quantum random oracle proofs are much harder than in the classical setting. In particular, lazy sampling (where we pick each output of the random oracle on demand when it is queried) does not work since all inputs can be queried simultaneously on the first query. Nonetheless, for a number of special cases, there are proof techniques for the quantum random oracle model (QROM). For example, $2q$-wise functions [47], semi-constant distributions [47], small range distribution [46], oracle indistinguishability

[46], and the one-way to hiding (O2H) theorem [44], just to mention a few.[1]

Of importance for this paper is the O2H theorem which exists in a number of variants (e.g., [1, 40, 41, 44]). The O2H theorem tells us – very roughly – that the probability of noticing whether a specific output $H(x)$ of the random oracle has been changed ("reprogrammed") can be bounded in terms of the probability of guessing that input x. This technique is used in a number of QROM proofs, in particular those for the FO transform described next.

Fujisaki-Okamoto. A common approach for constructing public key encryption schemes is the Fujisaki-Okamoto (FO) transform [18] or a variant thereof. The FO transform takes a public-key encryption scheme with some weak passive security notion (such as IND-CPA or one-way security) and transforms it into an actively secure public-key encryption or KEM[2] scheme (IND-CCA security). On a very high level, instead of executing the encryption algorithm with true randomness, the FO transform hashes the plaintext and uses the resulting hash value as the randomness for the encryption algorithm. This removes some of the flexibility the attacker has when constructing fake ciphertexts and makes chosen-ciphertext attacks impossible. The advantage of the FO transform is that it gets us IND-CCA security at no or almost no increase in ciphertext size or computational cost. The disadvantage is that the FO transform is only proven in the random oracle model, which means that there is a heuristic element to its security proof. Due to its high efficiency, the FO transform or some variations thereof is used in basically all public key encryption candidates in the NIST competition. Because of this, it is very important to understand the post-quantum security of the FO transform. However, due to the intricacies of the quantum random oracle model, proving the security of the FO transform is not as easy as in the classical setting. The first positive result was made by Ebrahimi Targhi and Unruh [36] who proved the security of an FO variant that includes one more hash value in the ciphertext. This additional hash had no obvious advantages but was required for their proof technique.[3] That result was adapted by [21] to several other FO variants, but still using an additional hash. ([21] also gives an excellent overview over the different FO variants, even if the security results are not state-of-the-art any more.) The first result to prove post-quantum security of FO without an additional hash was given by Saito, Xagawa, and Yamakawa [32]. To achieve this, they introduced a new intermediate security notion called "disjoint simulatability". However, [32] relies on the assumption that the underlying passively-secure encryption scheme has perfect correctness, i.e., the probability of incorrectly decrypting honestly generated ciphertexts is zero. Unfortunately, this is not the case with typical lattice-based encryption schemes (they have a negligible failure probability), making the results of [32] inapplicable to many relevant encryption schemes such as, to the best of our knowledge, all lattice-based NIST candidates. This situation was resolved by Hövelmanns, Kiltz, Schäge, and Unruh [23] who show the security of an FO variant (without additional hash) that is secure even in the presence of decryption failures. (This result is the one we formalize in this work. We will refer to [23], specifically to the part concerned with the FO transformation, as HKSU in the following.)

Formal verification of cryptography. As mentioned above, a state-of-the-art approach for writing cryptographic security proofs are sequences of games. This approach is also well suited

[1] However, these do not and cannot cover all situations handled by classical proof techniques. Ambainis, Rosmanis, and Unruh [2] show that in some cases, protocols that are secure in the classical ROM are insecure in the QROM.

[2] A KEM, key encapsulation scheme, differs from an encryption scheme in that it does not allow us to encrypt an arbitrary message but instead encrypts a uniformly random message. Raw public key encryption schemes are almost always combined with symmetric encryption schemes to build hybrid encryption schemes. In these, the public key scheme is only used to encrypt a random key anyway, so a KEM is sufficient. (See [17] for more on this.) Hence KEMs and public key encryption schemes are equivalent for most practical purposes.

[3] Having the additional hash allows us to define an extraction algorithm that inverts the hash in order to recover the randomness used by the encryption algorithm. In the classical setting this is not necessary because we can simply observe the list of oracle queries performed by the adversary.

for formal verification. A number of frameworks/tools use this approach for verifying classical cryptography: CryptoVerif [12], CertiCrypt [6], EasyCrypt [4], FCF [30], CryptHOL [7], and Verypto [11]. CryptoVerif tries to automatically determine a sequence of games by using a set of fixed rewriting rules for games. This has the advantage of reducing user effort, but it also means that the framework is more limited in terms of what game transformations are possible. In contrast, the other frameworks require the user to explicitly specify the games that constitute the security proof (as is done in a pen-and-paper proof), and to additionally provide justification for the fact that two consecutive games are indeed related as claimed. This justification will often be considerably more detailed than in a pen-and-paper proof where the fact that two slightly different games are equivalent will often be declared to be obvious.

One approach for proving the relationship of consecutive games is to give a proof in relational Hoare logic. Relational Hoare logic is a logic that allows us to express the relationship between two programs by specifying a relational precondition and a relational postcondition. A relational Hoare judgment of the form $\{\mathsf{A}\}\,\mathfrak{c} \sim \mathfrak{d}\,\{\mathsf{B}\}$ intuitively means that if the variables of the programs $\mathfrak{c}$ and $\mathfrak{d}$ are related as described by the precondition A before execution, and we execute $\mathfrak{c}$ and $\mathfrak{d}$, then afterwards their variables will be related as described by B. A very simple example would be $\{x_1 \leq x_2\}\, x \leftarrow x+1 \sim x \leftarrow x+1\, \{x_1 \leq x_2\}$. This means that if the variable x in the left program is smaller-equal than in the right one, and both programs increase x, then x in the left program will still be smaller-equal than in the right one. As this example shows, relational Hoare logic can express more complex relationships than simple equivalence of two games. This makes the approach very powerful. To reason about cryptography, one needs a variant of relational Hoare logic that supports probabilistic programs. Such a probabilistic relational Hoare logic (pRHL) was developed for this purpose by Barthe, Grégoire, and Zanella Béguelin [6]. Both CertiCrypt [6] and its popular successor EasyCrypt use pRHL for proving the relationship between cryptographic games.

Formal verification of quantum cryptography. When considering the verification of post-quantum cryptography, one might wonder whether the tools developed for classical cryptography may not already be sufficient.[4] Unfortunately, this is not the case. The soundness of the existing tools is proven relative to classical semantics of the protocols and of the adversary. In fact, at least for EasyCrypt and CryptHOL, Unruh [43] gave an explicit example of a protocol[5] which can be shown secure in EasyCrypt and CryptHOL but which is known not to be secure against quantum adversaries. For the purpose of verifying quantum cryptography, Unruh [43] introduced a generalization of pRHL, quantum relational Hoare logic (qRHL) that allows to prove relational statements about quantum programs. (We will describe qRHL in more detail in Section 2.) Unruh [43] also developed a tool `qrhl-tool` for reasoning in qRHL for the purpose of verifying quantum cryptography. However, except for a toy example, the post-quantum security of a very simple encryption scheme[6], to the best of our knowledge, no post-quantum security proof has been formally verified before this work. `qrhl-tool` uses a hybrid approach: Reasoning about qRHL judgments is hardcoded in the tool, but verification conditions (i.e., auxiliary subgoals, e.g., implications between invariants) are outsourced to the theorem prover Isabelle/HOL [28].

Our contribution. In this work, we formally verified the security proof of the FO transformation from HKSU [23].[7] The FO-variant analyzed by HKSU is a state-of-the-art construction for

[4] For actual quantum protocols where honest participants use quantum communication, this is clearly not the case because the protocol itself cannot be described in the tool.

[5] The CHSH game [16] which is a simple case of a multi-prover proof system.

[6] One-time IND-CPA security of the symmetric encryption scheme $\mathsf{Enc}(k, m) := G(k) \oplus m$.

[7] To be precise, we formalize the security proof from the February 2019 version [22] of [23]. The proof has been improved upon in later revisions of the paper. In particular, the requirement of injective encryption (see footnote 13) has been removed. We formalized the earlier version of the proof since the formalization was already under way when the proof was updated. Their new proof does not use substantially different techniques, and

building efficient public-key encryption schemes, and can be applied to many of the NIST submissions to get IND-CCA secure encryption schemes (e.g., Frodo [26] or Kyber [14]).

Our formalization follows the overall structure of HKSU (i.e., it uses roughly the same games) but introduces many additional intermediate games. (Altogether, our proof defines 136 programs, which covers games, oracles, and explicitly constructed adversaries in reductions.) The formalization has 3455 lines of proof in qRHL and 1727 lines of proof in Isabelle/HOL for auxiliary lemmas. (Not counting comments and blank lines or files autogenerated by search & replace from others.) We mostly follow the structure of HKSU (but in many places we need to do some adjustments to achieve the level of rigor required for formal verification). In the process, we identified a few best practices for doing proofs in `qrhl-tool` that we list in Section 2.4.

We furthermore extended the `qrhl-tool` with a tactic `o2h` that implements an application of the Semiclassical O2H Theorem [1]. This is needed in HKSU, but the O2H Theorem is often in post-quantum crypto proofs, so we expect this addition to be very useful for future verifications, too. (See Appendix B for details.)

Organization. In Section 2, we review qRHL and the `qrhl-tool`. In Section 3, we review the result and part of the proof from HKSU. In Section 4, we go through the parts of the formalization that make up the specification of the main result. In Section 5, we discuss the formal proof. The source code of the formalization is provided in [37].

2 Quantum Relational Hoare Logic

In this section, we give an overview of quantum relational Hoare logic (qRHL). We will not give formal definitions or a set of reasoning rules. For these, refer to [43]. Instead, our aim is to give an intuitive understanding of the logic that allows to understand (most of) the reasoning steps in our formalization.

2.1 Quantum While Language

qRHL allows us to reason about the relationship between quantum programs (that encode cryptographic games). The programs are written in a simple while language that has the following syntax (where $\mathfrak{c}$ and $\mathfrak{d}$ stand for programs):

$\mathfrak{c}, \mathfrak{d} := \mathbf{skip}$	(no operation)
$\mathfrak{c}; \mathfrak{d}$	(sequential composition)
$\mathbf{x} \leftarrow e$	(classical assignment)
$\mathbf{x} \xleftarrow{\$} e$	(classical sampling)
$\mathbf{if}\ e\ \mathbf{then}\ \mathfrak{c}\ \mathbf{else}\ \mathfrak{d}$	(conditional)
$\mathbf{while}\ e\ \mathbf{do}\ \mathfrak{c}$	(loop)
$\mathbf{q}_1 \dots \mathbf{q}_n \xleftarrow{\mathbf{q}} e$	(initialization of quantum registers)
$\mathbf{apply}\ e\ \mathbf{to}\ \mathbf{q}_1 \dots \mathbf{q}_n$	(quantum application)
$\mathbf{x} \leftarrow \mathbf{measure}\ \mathbf{q}_1 \dots \mathbf{q}_n\ \mathbf{with}\ e$	(measurement)
$\{\mathbf{local}\ V; \mathfrak{c}\}$	(local variable declaration)

The language distinguishes two kinds of variables, quantum and classical. In the above syntax, classical variables are denoted by $\mathbf{x}$ and quantum variables by $\mathbf{q}$. The command $\mathbf{x} \leftarrow e$ evaluates the expression e (which can be any well-typed mathematical expression involving only classical

we believe that formalizing their new proof in qRHL would not pose any challenges different from the ones encountered in this work. However, since their new proof is an almost complete rewrite (i.e., a different proof), it is not possible to simply update our formalization. Instead, a new development from scratch would be needed.

variables) and assigns the value to $\mathbf{x}$. In contrast $\mathbf{x} \overset{\$}{\leftarrow} e$ evaluates e which is supposed to evaluate to a distribution $\mathcal{D}$, and then samples the new value of $\mathbf{x}$ according to $\mathcal{D}$. If- and while-statements branch depending on a classical expression e.

To initialize quantum variables, we use $\mathbf{q}_1 \dots \mathbf{q}_n \overset{\mathbf{q}}{\leftarrow} e$. Here e is evaluated to return a quantum state (i.e., a classical expression returning the description of a quantum state). Then $\mathbf{q}_1 \dots \mathbf{q}_n$ are jointly initialized with that state. E.g., we can write $\mathbf{q} \overset{\mathbf{q}}{\leftarrow} |\mathbf{x}\rangle$ (where $\mathbf{x}$ is a classical bit variable) to initialize a quantum bit $\mathbf{q}$.

Given an expression e that computes a unitary U, we use **apply** e **to** $\mathbf{q}_1 \dots \mathbf{q}_n$ to apply U jointly to $\mathbf{q}_1 \dots \mathbf{q}_n$. E.g., **apply** CNOT **to** $\mathbf{q}_1\mathbf{q}_2$.

$\mathbf{x} \leftarrow$ **measure** $\mathbf{q}_1 \dots \mathbf{q}_n$ **with** e evaluates e to get a description of a measurement, measures $\mathbf{q}_1 \dots \mathbf{q}_n$ jointly with that measurement and assigns the result to $\mathbf{x}$. Typically, e might be something like `computational_basis`, denoting a computational basis measurement.

Finally, $\{$**local** $V; \mathfrak{c}\}$ declares the variables V as local inside $\mathfrak{c}$. (This is an extension of the language from [39].)

2.2 qRHL Judgements

Recall from the introduction that in relational Hoare logics, judgments are of the form $\{\mathsf{A}\}\,\mathfrak{c} \sim \mathfrak{d}\,\{\mathsf{B}\}$ where $\mathfrak{c}, \mathfrak{d}$ are programs, and A, B are relational predicates (the pre- and postcondition). In particular, $\{\mathsf{A}\}\,\mathfrak{c} \sim \mathfrak{d}\,\{\mathsf{B}\}$ means that if the variables of $\mathfrak{c}, \mathfrak{d}$ (jointly) satisfy A before execution, then they (jointly) satisfy B after execution.[8]

Predicates. The same idea is used in qRHL but the concept of predicates becomes more complex because we want to express something about the state of quantum variables. In fact, predicates in qRHL are subspaces of the Hilbert space of all possible states of the quantum variables of the two programs. We will illustrate this by an example:

Say $\mathbf{q}$ is a quantum variable in the first program $\mathfrak{c}$. We refer to $\mathbf{q}$ as $\mathbf{q}_1$ to distinguish it from a variable with the same name in program $\mathfrak{d}$. Say we want to express the fact that $\mathbf{q}_1$ is in state $|+\rangle$. That means that the whole system (i.e., all quantum variables together), are in a state of the form $|+\rangle_{\mathbf{q}_1} \otimes |\Psi\rangle_{\mathbf{vars}}$ where **vars** are all other variables (of $\mathfrak{c}$ and $\mathfrak{d}$), except $\mathbf{q}_1$. The set of all states of this form forms a subspace of the Hilbert space of all possible states of the quantum variables. Thus $\mathsf{A} := \{|+\rangle_{\mathbf{q}_1} \otimes |\Psi\rangle_{\mathbf{vars}} : |\Psi\rangle \in \mathcal{H}_{\mathbf{vars}}\}$ (with $\mathcal{H}_{\mathbf{vars}}$ denoting the corresponding Hilbert space) is a subspace and thus a valid predicate for use in a qRHL judgment. For example, we could then write $\{\mathsf{A}\}\,$**apply** X **to** $\mathbf{q} \sim$ **skip** $\{\mathsf{A}\}$ to express the fact that, if $\mathbf{q}$ is in state $|+\rangle$ in the left program, and we apply X (a quantum bit flip) to $\mathbf{q}$, then afterwards $\mathbf{q}$ is still in state $|+\rangle$. Of course, writing A explicitly as a set is very cumbersome. (And, in the setting of formal verification, one would then have no syntactic guarantees that the resulting set is indeed a valid predicate.) Instead, we have the shorthand $\mathrm{span}\{|+\rangle\}\text{»}\mathbf{q}_1$ to denote the above predicate A. (More generally, $S\text{»}\mathbf{q}_1 \dots \mathbf{q}_n$ means that $\mathbf{q}_1 \dots \mathbf{q}_n$ are jointly in a state in S.)

We can build more complex predicates by combining existing ones. E.g., if A, A' are predicates, then $\mathsf{A} \cap \mathsf{B}$ is a predicate that intuitively denotes the fact that both A and B hold. And $\mathsf{A} + \mathsf{B}$, the sum of two subspaces is somewhat akin to a disjunction of predicates. Or given an operator D (e.g., a unitary or a projector), $D \cdot \mathsf{A}$ is a predicate, too. ($D \cdot \mathsf{A}$ is the pointwise multiplication of the vectors in A with D.) These predicates have less clear intuitive meanings but may arise from applying certain reasoning rules.

We will also often have to compare predicates. $\mathsf{A} \subseteq \mathsf{B}$ means that A is a subspace of B for all values of the classical variables. Intuitively, this means A implies B.

[8] There is a subtlety here: Since the two programs are not jointly executed (they live in separate worlds, so to say) there is no obvious notion of the joint distribution or state of the programs after execution. Thus the definition of the meaning of $\{\mathsf{A}\}\,\mathfrak{c} \sim \mathfrak{d}\,\{\mathsf{B}\}$ is a bit more complicated than presented here. We refer to [43] for a formal exposition.

Predicates with classical variables. In most cases, however, we do not only have quantum variables, but also classical variables. Especially in a post-quantum cryptography setting, the majority of variables in a predicate tends to be classical. Support for classical variables in qRHL predicates is straightforward: A predicate A can be an expression containing classical variables. Those are then substituted with the current values of those variables, and the result is a subspace that describes the state of the quantum variables as explained above. For example, we can write the predicate $\mathrm{span}\{|\mathbf{x}_2\rangle\}\text{»}\mathbf{q}_1$. This would mean that $\mathbf{q}_1$ (a qubit in the left program) is in state $|\mathbf{x}_2\rangle$.

This already allows us to build complex predicates, but it is rather inconvenient if we want to express something about the classical variables only, e.g., if we want to express that $\mathbf{x}_1 = \mathbf{x}_2$ always holds. To express such classical facts, we use an additional shorthand: $\mathfrak{Cla}[b]$ is defined to be $\mathcal{H}$ (the full Hilbert space) if $b = \mathtt{true}$, and defined to be 0 (the subspace containing only 0) if $b = \mathtt{false}$. Why is this useful? Consider the predicate $\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{x}_2]$. If $\mathbf{x}_1 = \mathbf{x}_2$, this evaluates to $\mathfrak{Cla}[\mathtt{true}] = \mathcal{H}$. Since $\mathcal{H}$ contains all possible states, the state of the quantum variables will necessarily be contained in $\mathfrak{Cla}[\mathtt{true}]$, hence the predicate is satisfied. If $\mathbf{x}_1 \neq \mathbf{x}_2$, $\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{x}_2]$ evaluates to $\mathfrak{Cla}[\mathtt{false}] = 0$, and the state of the quantum variables will not be contained in $\mathfrak{Cla}[\mathtt{false}]$, hence the predicate will not be satisfied. Thus, $\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{x}_2]$ is satisfied iff $\mathbf{x}_1 = \mathbf{x}_2$; the state of the quantum variables does not matter. In general $\mathfrak{Cla}[e]$ allows us to translate any classical predicate e into a quantum predicate. (And this predicate can then be combined with quantum predicates, e.g., $\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{x}_2] \cap \mathrm{span}\{|+\rangle\}\text{»}\mathbf{q}_1$.)

Quantum equality. One very important special case of predicates are equalities. We will often need to express that the variables of the left and right programs have the same values. We have already seen how to do this for classical variables. For quantum variables, the situation is more complex. We cannot write $\mathfrak{Cla}[\mathbf{q}_1 = \mathbf{q}_2]$, this is not even a meaningful expression (inside $\mathfrak{Cla}[\ldots]$, only classical variables are allowed). Instead, we need to define a *subspace* that in some way expresses the fact that two quantum variables are equal. The solution proposed in [43] (and that is shown there to be the only possibility if we expect certain natural laws to hold) is: Let $\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2$ denote the subspace of all states that are invariant under exchanging $\mathbf{q}_1$ and $\mathbf{q}_2$ (i.e., invariant under applying a swap operation). Then $\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2$ is a quantum predicate (but not a Boolean expression, notwithstanding the suggestive notation). And – this is less easy to see but shown in [43] – $\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2$ does indeed capture the idea that $\mathbf{q}_1$ and $\mathbf{q}_2$ have the same value in a meaningful way. We can now write, for example, $\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{x}_2] \cap (\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2)$ to denote the fact that the variables $\mathbf{x}$ (classical) and $\mathbf{q}$ (quantum) have the same value in both programs. In particular, if $\mathfrak{c}$ only contains those two variables, we have $\big\{\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{x}_2] \cap (\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2)\big\}\,\mathfrak{c} \sim \mathfrak{c}\,\big\{\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{x}_2] \cap (\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2)\big\}$. What if there are more quantum variables? The advantage of the quantum equality is that we hardly ever need to recall the actual definition in terms of swap invariance. All we need to remember is that $\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2$ is a quantum predicate/subspace that intuitively encodes equality of $\mathbf{q}_1$ and $\mathbf{q}_2$.

If we have more than two variables that we wish to compare, $\mathbf{q}_1^{(1)} \ldots \mathbf{q}_1^{(n)} \equiv_{\mathsf{quant}} \mathbf{q}_2^{(1)} \ldots \mathbf{q}_2^{(n)}$ can be defined analogously. It intuitively means that the variable tuple $\mathbf{q}_1^{(1)} \ldots \mathbf{q}_1^{(n)}$ in the first program has the same state as $\mathbf{q}_2^{(1)} \ldots \mathbf{q}_2^{(n)}$ in the second program.

There are, however, a few pitfalls and subtleties that one should be aware of:

- While $\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_1'$ (comparison of two variables in the left program) is well-defined, it lacks a clear intuitive meaning. The idea that quantum equality implies the equality of the states of the variables only works if all the variables on the lhs (left hand side) of $\equiv_{\mathsf{quant}}$ are from the left program, and all the variables on the rhs (right hand side) are from the right one. (Or vice versa.)
- $\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2$ implies more than just equality between $\mathbf{q}_1$ and $\mathbf{q}_2$. It additionally means that $\mathbf{q}_1$ and $\mathbf{q}_2$ are not entangled with any other variables. There is no way to express equality between variables that may be entangled with other variables not occurring in the

quantum equality.

- While classical equality satisfies the rule that $(\mathbf{x}_1, \mathbf{y}_1) = (\mathbf{x}_2, \mathbf{y}_2)$ is equivalent to $\mathbf{x}_1 = \mathbf{x}_2 \wedge \mathbf{y}_1 = \mathbf{y}_2$, this is not the case for quantum equality. In fact, $(\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2) \cap (\mathbf{q}_1' \equiv_{\mathsf{quant}} \mathbf{q}_2')$ implies (is a subset of) $\mathbf{q}_1\mathbf{q}_1' \equiv_{\mathsf{quant}} \mathbf{q}_2\mathbf{q}_2'$ but not vice versa. (This should not come as a total surprise given the previous point: $(\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2) \cap (\mathbf{q}_1' \equiv_{\mathsf{quant}} \mathbf{q}_2')$ implies that $\mathbf{q}_1$ and $\mathbf{q}_2$ are not entangled, while $\mathbf{q}_1\mathbf{q}_1' \equiv_{\mathsf{quant}} \mathbf{q}_2\mathbf{q}_2'$ allows them to be entangled with each other, just not with other variables.)

The most common form of predicate that we will see is $\mathsf{A}_= := \mathfrak{Cla}[\mathbf{x}_1^{(1)} = \mathbf{x}_2^{(1)} \wedge \cdots \wedge \mathbf{x}_1^{(1)} = \mathbf{x}_2^{(1)}] \cap (\mathbf{q}_1^{(1)} \dots \mathbf{q}_1^{(m)} \equiv_{\mathsf{quant}} \mathbf{q}_2^{(1)} \dots \mathbf{q}_2^{(m)})$. In fact, if both sides have the same program $\mathfrak{c}$ (and $\mathfrak{c}$ contains no variables besides the ones mentioned in $\mathsf{A}_=$), then $\{\mathsf{A}_=\}\,\mathfrak{c} \sim \mathfrak{c}\,\{\mathsf{A}_=\}$ holds. Intuitively, this means: if the inputs of two programs are equal, the outputs are equal.

2.3 Reasoning in qRHL

To derive qRHL judgments, one will hardly ever go directly through the definition of qRHL itself. Instead one derives complex qRHL judgments from elementary ones. For example, to derive the elementary $\{\mathfrak{Cla}[\mathbf{x}_1 = 0]\}\,\mathbf{x} \leftarrow \mathbf{x} + 1 \sim \mathbf{skip}\,\{\mathfrak{Cla}[\mathbf{x}_1 = 1]\}$, we use the Assign1 rule [43] that states $\big\{\mathsf{B}\{e_1/\mathbf{x}_1\}\big\}\,\mathbf{x} \leftarrow e \sim \mathbf{skip}\,\big\{\mathsf{B}\big\}$. (Here e_1 is the expression e where all variables $\mathbf{y}$ are replaced by $\mathbf{y}_1$. And $\mathsf{B}\{e_1/\mathbf{x}_1\}$ means every occurrence of $\mathbf{x}_1$ in B is replaced by e_1.) With $\mathsf{B} := \mathfrak{Cla}[\mathbf{x}_1 = 1]$, we get from Assign1: $\big\{\mathfrak{Cla}[\mathbf{x}_1 + 1 = 1]\big\}\,\mathbf{x} \leftarrow e \sim \mathbf{skip}\,\big\{\mathfrak{Cla}[\mathbf{x}_1 = 1]\big\}$. Since $\mathbf{x}_1 + 1 = 1$ is logically equivalent to $\mathbf{x}_1 = 1$ (assuming the type of $\mathbf{x}$ is, e.g., integers or reals), this statement is equivalent to $\{\mathfrak{Cla}[\mathbf{x}_1 = 0]\}\,\mathbf{x} \leftarrow \mathbf{x} + 1 \sim \mathbf{skip}\,\{\mathfrak{Cla}[\mathbf{x}_1 = 1]\}$. (This is an example of reasoning in the "ambient logic": Besides application of qRHL rules, we need to use "normal" mathematics to derive facts about predicates. This is external to qRHL itself.)

One can then combine several judgments into one, using, e.g., the Seq rule: "If $\{\mathsf{A}\}\,\mathfrak{c} \sim \mathfrak{d}\,\{\mathsf{B}\}$ and $\{\mathsf{B}\}\,\mathfrak{c}' \sim \mathfrak{d}'\,\{\mathsf{C}\}$ holds, then $\{\mathsf{A}\}\,\mathfrak{c};\mathfrak{c}' \sim \mathfrak{d};\mathfrak{d}'\,\{\mathsf{C}\}$ holds." For example, once we have derived $\{\mathfrak{Cla}[\mathtt{true}]\}\,\mathbf{x} \leftarrow 1 \sim \mathbf{skip}\,\{\mathfrak{Cla}[\mathbf{x}_1 = 1]\}$ and $\{\mathfrak{Cla}[\mathbf{x}_1 = 1]\}\,\mathbf{skip} \sim \mathbf{y} \leftarrow 1\,\{\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{y}_2]\}$ (using the Assign1 rule and its rhs variant Assign2), we conclude using Seq that $\{\mathfrak{Cla}[\mathtt{true}]\}\,\mathbf{x} \leftarrow 1 \sim \mathbf{y} \leftarrow 1\,\{\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{y}_2]\}$. (We use here implicitly that $\mathbf{x} \leftarrow 1; \mathbf{skip}$ is the same as $\mathbf{x} \leftarrow 1$ and analogously for $\mathbf{skip}; \mathbf{y} \leftarrow 1$.)

We will not give the full list of rules here, see [43] and the manual of [38] for a comprehensive list.

One common approach to prove more complex qRHL judgments is backward reasoning: One starts by stating the judgment one wants to prove, say $G_1 := \{\mathfrak{Cla}[\mathtt{true}]\}\,\mathbf{x} \leftarrow 1 \sim \mathbf{y} \leftarrow 1\,\{\mathfrak{Cla}[\mathbf{x}_1 = \mathbf{y}_2]\}$. Then one applies one qRHL rule to the very last statement on the left or right, say $\mathbf{y} \leftarrow 1$. By application of the Seq and Assign2 rule, we see that $G_2 := \{\mathfrak{Cla}[\mathtt{true}]\}\,\mathbf{x} \leftarrow 1 \sim \mathbf{skip}\,\{\mathfrak{Cla}[\mathbf{x}_1 = 1]\}$ implies G_1. So we have reduced our current goal to showing G_2. (Using a reasoning step that can be fully automated.) By application of Seq and Assign1, we see that $G_3 := \{\mathfrak{Cla}[\mathtt{true}]\}\,\mathbf{skip} \sim \mathbf{skip}\,\{\mathfrak{Cla}[1 = 1]\}$ implies G_2. So our new proof goal is G_3. And finally, G_3 is implied by $G_4 := \big(\mathfrak{Cla}[\mathtt{true}] \subseteq \mathfrak{Cla}[1 = 1]\big)$. So our final goal is G_4 which is a trivial statement in ambient logic because $1 = 1$ is `true`. Hence the proof concludes and G_1 holds. The advantage of this approach is that it is fully mechanical in many cases, e.g., for sequences of assignments and applications of unitaries. The proof tool `qrhl-tool` (see the next section) follows exactly this approach.

So far, we have gotten a glimpse how to derive qRHL judgments. In a cryptographic proof, however, we are interested not in qRHL judgments but in statements about probabilities. Fortunately, we can derive those directly from a qRHL judgment using the QrhlElimEq rule. It states (somewhat simplified): Assuming $\mathbf{X}$ and $\mathbf{Q}$ are all the variables occurring in $\mathfrak{c}, \mathfrak{d}$, then $\big\{\mathfrak{Cla}[\mathbf{X}_1 = \mathbf{X}_2] \cap (\mathbf{Q}_1 \equiv_{\mathsf{quant}} \mathbf{Q}_2)\big\}\,\mathfrak{c} \sim \mathfrak{d}\,\big\{\mathfrak{Cla}[e_1 \implies f_2]\big\}$ implies $\Pr[e : \mathfrak{c}] \leq \Pr[f : \mathfrak{d}]$. (Here $\Pr[e : \mathfrak{c}]$ denotes the probability that the Boolean expression e is true after executing $\mathfrak{c}$, and analogously $\Pr[f : \mathfrak{d}]$.) A variant of the rule (with $\iff$ instead of $\implies$) allows us to derive an

equality of probabilities. Thus to derive an inequality or equality of probabilities of program outcomes, we convert it into a qRHL proof goal with QrhlElimEq, and then use the reasoning rules of qRHL to derive that qRHL judgment. This is, on a high level, how crypto proofs in qRHL are done (modulo many concrete details).

2.4 The `qrhl-tool`

While reasoning using qRHL in pen-and-paper proofs is possible in principle, qRHL was specifically designed for formal verification on the computer. To that end, an interactive theorem prover for qRHL was developed, `qrhl-tool` [38, 43]. To execute our formalization, version 0.5 is required. See README.txt there for instructions on how to check/edit our formalization, and manual.pdf for detailed information. In the following, we recap the most important facts about the tool.

In addition to that review, we also list some "best practices" for developing proofs in the tool, based on our experience while formalizing HKSU.

Architecture of the tool. `qrhl-tool` has a hybrid architecture: It embeds the theorem prover Isabelle/HOL, and all reasoning in the ambient logic is done by Isabelle/HOL. The tool handles qRHL judgments directly. As a consequence, proofs are written in two files: `.thy` files contain native Isabelle/HOL code and can reason only about ambient logic (no support for qRHL itself). Those `.thy` files are also used to specify the logical background of the formalization (e.g., declaring constants such as the encryption function in our development). `.qrhl` files are executed natively by `qrhl-tool` and contain specifications of programs, as well as proofs in qRHL. They can also contain proofs in ambient logic (arbitrary Isabelle/HOL tactics can be invoked) but this is only suitable for simple proofs in ambient logic. Complex ambient logic facts are best proven as an auxiliary lemma in the `.thy` files. It is possible to split a proof into many files by including one `.qrhl` file from another using the `include` command.[9]

The tool can be run in two modes, batch and interactive mode. In batch mode, a given `.qrhl` file is checked on the command-line and the run aborts with an error if the proof is incorrect. In interactive mode, an Emacs/ProofGeneral-based user interface allows us to edit and execute the proofs. In this mode, included files are not checked, this speeds up the development process significantly.

Best practice: *Create one file `variables.qrhl` that declares all program variables and loads the `.thy` files. Furthermore, create a separate file `p.qrhl` for every declared program `p`, and a separate file `lemma_l.qrhl` for every lemma `l`. This allows us to execute proofs without too much runtime overhead and at the same time allows us to find quickly which entity is declared where.* ◇

Declarations. All program variables that occur in any analyzed program need to be declared globally (even if the variable is used only as a local variable). This is done using `classical/quantum/ambient var x : type` where `type` is any type understood by Isabelle/HOL. (`ambient var` declares an unspecified constant value that can be used in programs and in ambient logic subgoals.) Programs are declared by `program name := { code }` where `code` is a quantum program as described in Section 2.1. For describing games this approach is sufficient, but when specifying adversaries or oracles or helper functions, we would like to define procedures that take arguments and have return values. Unfortunately, such procedure calls are not supported by the language underlying qRHL yet. What has to be done instead is to pass values to/from

[9] The tool automatically disregards duplicate include commands, thus it is possible to have files with a complex dependency structure.

procedures via global variables. A program `X` can be invoked by another program using `call X`,[10] and we need to write the program `X` so that it communicates with the invoking program via global variables that are set aside for this purpose. While this approach is not very convenient, we found that with disciplined variable use, no bigger problems arise.

One highly important feature in more advanced cryptographic definitions and proofs are oracles. Roughly speaking, an oracle is a program O that is given as an argument to another program A, so that the other program can execute it whenever needed. (For example, an adversary A may get access to a decryption oracle `Dec` that decrypts messages passed to it.) Programs that take oracles are supported by `qrhl-tool`. One can declare a program via, e.g., `program prog(O1,O2) := {code}` where `code` can contain, e.g., a `call O1` statement. Then `prog` is a program that needs to be instantiated with oracles when invoked, e.g.: `call prog(Enc,Dec)`.

Finally, to model adversaries we need to declare programs of unspecified code. (This then means that anything that is proven holds for any adversary.) The command `adversary A` declares an adversary `A` that can be invoked via `call A`. Additional arguments to the `adversary` command allow to specify global variables that the adversary may access, and whether `A` expects oracles.

BEST PRACTICE: *When declaring a program that is intended as a subroutine (e.g., an oracle or an adversary), make explicit which global variables are used as inputs/outputs to simulate procedure calls. (E.g., an adversary might be annotated with a comment "Input:* `c` *(ciphertext). Output:* `b` *(guessed bit). Internal state:* `stateA`*.")*

All variables (especially quantum variables) that are used that are not needed between consecutive invocations should be made local at the beginning *of the program using the* `local` *program statement.*

When invoking a program taking an oracle (e.g., `call A(queryH)` *where* `queryH` *expects inputs/outputs in variables* `Hin,Hout`*), the input/output variables should be made local at that call. (E.g.,* `{ local Hin,Hout; call A(queryH); }`*.) Otherwise,* `qrhl-tool` *will not be able to determine that* `Hin,Hout` *are not changed globally, even if the code of* `A` *internally already contains a* `local` *statement.*

`print programname` *can be used in interactive mode to check the list of variables used by a program.*

Following these rules may make many proofs somewhat longer (due to additional boilerplate for removing `local` *commands) but it removes a lot of potential for conflicts in variable use. (Especially with quantum variables: due to the idiosyncrasies of the quantum equality, any quantum variable used non-locally by a subprogram will have to be carried around explicitly in all quantum equalities.)* ◇

Note that qRHL did not initially support local variables. The addition of local variables to `qrhl-tool` and the corresponding theory [39] were prompted by our experiences in the present formalization. Without local variables, it becomes very difficult to maintain a larger formalization involving quantum equalities.

Proving lemmas. Finally, to state a lemma one can either state a lemma in ambient logic (extended with syntax `Pr[...]` for talking about the results of program executions), or qrhl subgoals. The command `lemma name: formula` states the former, the command `qrhl name: {precondition} call X; ~ call Y; {postcondition}` the latter. The syntax `Pr[e : prog(rho)]` denotes the probability that the Boolean expression `e` is true after running `prog` with initial quantum state `rho`. Most of the time we will thus state lemmas of the form `Pr[b=1 : prog1(rho)] = Pr[b=1 : prog2(rho)]` where `rho` is an ambient variable (meaning, the initial state is arbi-

[10] Semantically, `call X` is not a separate language feature. It just means that the source code of `X` is included verbatim at this point.

trary). This can be converted into a qRHL subgoal using the tactic `byqrhl` (implementing the rule QrhlElimEq).

Once one has stated a qRHL proof goal, the proof proceeds via backwards reasoning as described in Section 2.3. For example, to remove the last assign statement from a goal (and rewrite the postcondition accordingly) as done in Section 2.3, one writes `wp left/right` (or `wp` n m for the last n/m statements on the left/right). Once all statements are gone (`skip` on both sides), the tactic `skip` replaces $\{\mathsf{A}\}\,\mathbf{skip} \sim \mathbf{skip}\,\{\mathsf{B}\}$ by the ambient logic goal $\mathsf{A} \subseteq \mathsf{B}$. Another important tactic is `conseq pre/post:` C which replaces the current pre-/postcondition by C (and adds a ambient logic subgoal to prove that this replacement is justified). This allows to clean up subgoals and increases readability. The tactic `simp` simplifies the current goal using the Isabelle/HOL simplifier.

To operate on ambient logic subgoals, one can use, e.g., `simp` (which may remove a subgoal if it can be proven by the Isabelle/HOL simplifier). Or `rule` x to apply a lemma x that has been proven, e.g., in the `.thy` files. Or `isa` m to apply an arbitrary Isabelle/HOL proof method m. (The full syntax of Isabelle/HOL is supported in the latter. A useful simple example is `isa auto` which often works where `simp` fails.)

Once all subgoals are proven, `qed` finishes the proof and the lemma is available for use in other proofs.

Best practice: *To make proofs more maintainable, before each tactic invocation add a comment which line of code it addresses. E.g., `wp left` will always affect the last command of the left program. If that command is, e.g., `x <- 1+y`, add the comment `# x`. This ensures that if a change in a program definition breaks an existing proof, it is easier to find out where the proof script went out of sync.*

Additionally, at regular intervals add the tactic `conseq post:` `X` commands where `X` is the current postcondition (possibly, but not necessarily simplified). This serves both as a documentation of the current state of the proof, and it makes maintenance easier because the proof will fail at the first point where the postcondition is not what was expected anymore (as opposed to failing at a later point). ◇

Isabelle/HOL micro primer. For an introduction to Isabelle/HOL we recommend [27]. Here, we only give some minimal information to help reading the code fragments in the paper (Figures 7–9). This micro primer does not does not allow us to understand the definitions given in this paper in depth. In particular, to understand them in depth one needs to know the predefined constants in Isabelle/HOL and in the `qrhl-tool`. But with the syntax given here, it should at least be possible to make educated guesses about the meanings of the definitions.

All constants in Isabelle/HOL are typed. A function `f` taking arguments of types $t_1, \ldots, t_n$ and returning t has type $t_1 \Rightarrow \cdots \Rightarrow t_n \Rightarrow t$. To invoke `f` with arguments $a_1, \ldots, a_n$, we write `f` a_1 a_2 ... a_n. (Not `f`$(a_1, \ldots, a_n)$.) To declare (axiomatically) the existence of a new constant `c` of type *type*, we write

```
axiomatization c :: type where facts
```

Here the optional `facts` are logical propositions that we assume about `c`. For example, we can declare the existence of a commutative binary operation `op` on natural numbers via

```
axiomatization op :: "nat⇒nat⇒nat" where comm: "op x y = op y x"
```

Instead of axiomatizing constants, we can also define them in terms of existing constants. This cannot introduce logical inconsistencies. The syntax for this is

```
definition c :: type where "c arguments = definition"
```

Instead of = we can also write ↔ when defining a proposition (i.e., if the return type is `bool`). For example, if we wanted to define the operation `op` above as twice the sum of its arguments, we could write:

$\underline{\mathsf{DS}_{\mathsf{real}}}$
01 $(pk, sk) \leftarrow \mathsf{Keygen}()$
02 $m \xleftarrow{\$} \mathcal{M}$
03 $c \leftarrow \mathsf{Enc}(pk, m)$
04 $b \leftarrow A(pk, c)$

$\underline{\mathsf{DS}_{\mathsf{fake}}}$
05 $(pk, sk) \leftarrow \mathsf{Keygen}()$
06 $c \leftarrow \overline{\mathsf{Enc}}(pk)$
07 $b \leftarrow A(pk, c)$

Figure 1: Games from definition of disjoint simulatability. In the random oracle model, A is additionally given oracle access to all random oracles.

```
definition op :: "nat ⇒ nat ⇒ nat" where "op x y = 2 * (x + y)"
```

The parts before **where** are optional and will be inferred if necessary.

This summary of the operation of `qrhl-tool` does not, of course, replace a reading of the manual. However, it should give a first impression as well as help in reading Sections 4–5.

3 Fujisaki-Okamato à la HKSU

In this section, we describe the FO variant analyzed by HKSU [23] and their proof. We stress that the proof we analyzed (and describe here) is the one from the earlier version [22] of HKSU, it has been rewritten since we started our formalization.

The goal of the FO transform is to transform an encryption scheme that is passively secure into a chosen-ciphertext secure key encapsulation mechanism (KEM). The variant analyzed by HKSU can be described modularly by consecutively applying three transformations (called Punc, T, and $\mathsf{U}^{\not\perp}_m$) to the passively secure encryption scheme.

3.1 Transformation Punc

We start with a base public-key encryption scheme $(\mathsf{Keygen}_0, \mathsf{Enc}_0, \mathsf{Dec}_0)$ with message space $\mathcal{M}_0$. We assume the base scheme to be IND-CPA secure. (We assume further that decryption is deterministic, but we do not assume that decryption succeeds with probability 1, or that decrypting a valid ciphertext returns the original plaintext with probability 1.)

The first step is to construct a scheme with disjoint simulatability (DS). DS security [32] means that there exists a fake encryption algorithm $\overline{\mathsf{Enc}}$ that (without being given a plaintext) returns ciphertexts that are indistinguishable from valid encryptions of random messages, but that are guaranteed to be distinct from any valid ciphertext with high probability.

More precisely:

Definition 1 (Disjoint simulatability) *We call* $(\mathsf{Keygen}, \mathsf{Enc}, \mathsf{Dec})$ *with message space* $\mathcal{M}$ *and randomness space* $\mathcal{R}$ *DS secure iff for any quantum-polynomial-time* A, $\big|\Pr[b = 1 : \mathsf{DS}_{\mathsf{real}}] - \Pr[b = 1 : \mathsf{DS}_{\mathsf{fake}}]\big|$ *is negligible, for the games defined in Figure 1.*

We say $(\mathsf{Keygen}, \mathsf{Enc}, \mathsf{Dec})$ *is* ε*-disjoint iff for all possible public keys* pk, $\Pr\big[(\exists m \in \mathcal{M}, r \in \mathcal{R}.\ c = \mathsf{Enc}(pk, m; r)) : c \leftarrow \overline{\mathsf{Enc}}(pk)\big] \leq \varepsilon$.

The transformation Punc is very straightforward: The encryption scheme is not really modified (i.e., the resulting $\mathsf{Keygen}, \mathsf{Enc}, \mathsf{Dec}$ are the same in the base scheme). But the message space is reduced by one element. I.e., we simply declare one plaintext $\hat{m}$ as invalid, hence encryptions of that plaintext will be disjoint from valid ciphertexts, and thus we can produce fake encryptions $\overline{\mathsf{Enc}}$ by encrypting $\hat{m}$. We summarize Punc in Figure 2. (We slightly generalized the transformation by allowing $\mathcal{M}$ to be an arbitrary proper subset of $\mathcal{M}_0$, i.e., we can remove more than one element. This could be relevant if we want the set of messages to still be, e.g., representable as fixed length bitstrings.) The proof that the resulting scheme is both DS and IND-CPA secure is very straightforward, and we omit it here. (But we have formalized it, of course.)

$$\mathsf{Enc} := \mathsf{Enc}_0,\ \mathsf{Dec} := \mathsf{Dec}_0,\ \mathsf{Keygen} := \mathsf{Keygen}_0,$$
$$\mathcal{M} \subsetneq \mathcal{M}_0,\ \hat{m} \in \mathcal{M}_0 \setminus \mathcal{M},$$
$$\overline{\mathsf{Enc}}(pk) := \mathsf{Enc}_0(pk, \hat{m}).$$

Figure 2: Transformation Punc. Input scheme: $(\mathsf{Enc}_0, \mathsf{Dec}_0, \mathsf{Keygen}_0)$ with message space $\mathcal{M}_0$. Output scheme: $(\mathsf{Enc}, \mathsf{Dec}, \mathsf{Keygen})$ with message space $\mathcal{M}_0$ and fake encryption algorithm $\overline{\mathsf{Enc}}$.

$$\mathsf{Keygen}' := \mathsf{Keygen},\ \mathcal{M}' := \mathcal{M},\ \overline{\mathsf{Enc}'} := \overline{\mathsf{Enc}},$$
$$\mathsf{Enc}'(pk, m) := \mathsf{Enc}\big(pk, m; G(m)\big).$$
$$\mathsf{Dec}'(sk, c) := \textbf{if } m = \bot \textbf{ or } \mathsf{Enc}'(pk, m) \neq c$$
$$\textbf{then return } \bot \textbf{ else return } m$$
$$(\textbf{where } m := \mathsf{Dec}(sk, c)).$$

Figure 3: Transformation T. Input scheme: $(\mathsf{Enc}, \mathsf{Dec}, \mathsf{Keygen})$ with message space $\mathcal{M}$ and fake encryption algorithm $\overline{\mathsf{Enc}}$. Output scheme: $(\mathsf{Enc}', \mathsf{Dec}', \mathsf{Keygen}')$ with message space $\mathcal{M}'$ and fake encryption algorithm $\overline{\mathsf{Enc}'}$. $G : \mathcal{M} \to \mathcal{R}$ is a hash function (modeled as a random oracle).

3.2 Transformation T

Transformation Punc gave us a DS secure encryption scheme Enc. However, as the starting point for the transformation $\mathsf{U}^{\not\perp}_m$ below, we need a deterministic encryption scheme (that is still DS secure).

Transformation T achieves this by a simple technique: Instead of running Enc normally (i.e., $\mathsf{Enc}(pk, m; r)$ with r uniformly from the randomness space $\mathcal{R}$), the modified encryption algorithm Enc' computes the randomness r from the message m as $r := G(m)$. Here G is a hash function (modeled as a random oracle in the proof).

Transformation T also strengthens the decryption algorithm: The decryption algorithm resulting from T rejects any invalid ciphertexts (i.e., any ciphertext that is not in the range of Enc'). This is achieved by adding an extra check to the decryption algorithm Dec': After decrypting a ciphertext c, resulting in a message m, m is reencrypted and compared with the ciphertext c. Since Enc' is deterministic, this will always succeed for honestly generated ciphertexts, but it will always fail for invalid ones.

We summarize transformation T in Figure 3.

Security of transformation T. HKSU shows the following:

Theorem 1 (DS security of Enc', informal) *If Enc is ε-disjoint, so is Enc'. If Enc is DS secure and IND-CPA secure, then Enc' is DS secure.*

(In HKSU, the result is given with concrete security bounds.)

The core idea of the proof is to show that the adversary cannot distinguish between uniform randomness (as used in Enc) and randomness $r := G(m^*)$ where m^* is the challenge message (as used in Enc'). This is shown by bounding the probability for guessing m^* and then using the Semiclassical O2H theorem [1] to bound the distinguishing probability.

We omit the proof from this exposition (our explanations will focus on the more complex proof of transformation $\mathsf{U}^{\not\perp}_m$ below). The full proof can be found in [22].

3.3 Transformation $\mathsf{U}^{\not\perp}_m$

Finally, the transformation $\mathsf{U}^{\not\perp}_m$ takes the deterministic DS secure encryption scheme and transforms it into a KEM. In a KEM, we have an encapsulation algorithm Encaps that, instead of

$\underline{\mathsf{Keygen}_{\mathsf{FO}}()}$:
01 $(pk, sk') \leftarrow \mathsf{Keygen}'()$
02 $k \xleftarrow{\$} \mathcal{K}_{\mathsf{PRF}}$
03 $sk := (sk', k)$
04 **return** (pk, sk)

$\underline{\mathsf{Encaps}(pk)}$:
05 $m \xleftarrow{\$} \mathcal{M}'$
06 $c \leftarrow \mathsf{Enc}'(pk, m)$
07 $K := H(m)$
08 **return** (K, c)

$\underline{\mathsf{Decaps}(sk, c) \text{ with } sk = (sk', k)}$:
09 $m := \mathsf{Dec}'(sk', c)$
10 **if** $m = \bot$ **then return** $K := \mathsf{PRF}(k, c)$
11 **else return** $K := H(m)$

Figure 4: Transformation $\mathsf{U}^{\not\perp}_m$. Input scheme: $(\mathsf{Enc}', \mathsf{Dec}', \mathsf{Keygen}')$ with message space $\mathcal{M}'$ and fake encryption algorithm $\overline{\mathsf{Enc}'}$. Output scheme: $(\mathsf{Keygen}_{\mathsf{FO}}, \mathsf{Encaps}, \mathsf{Decaps})$ with key space $\mathcal{K}$. (The key space is the space of encapsulated keys, not of public/secret key pairs.) PRF is a pseudorandom function with key space $\mathcal{K}_{\mathsf{PRF}}$. $H : \mathcal{M}' \to \mathcal{K}$ is a hash function (modeled as a random oracle).

accepting a plaintext as input, uses a random (symmetric) key K as plaintext (intended for use in a symmetric encryption scheme) and returns both that key and the ciphertext. (We stress that K must not be confused with the public/secret keys of the KEM.) And the decapsulation algorithm Decaps takes the ciphertext and returns the key, like a decryption does.

The encapsulation algorithm constructed by transformation $\mathsf{U}^{\not\perp}_m$ picks a uniform $m \xleftarrow{\$} \mathcal{M}'$ and encrypts it (resulting in a ciphertext c). However, instead of using m directly as the symmetric key, the key is set to be $K := H(m)$. (Here H is a hash function modeled as a random oracle.)

Decapsulating c is straightforward: By decrypting c we get m back, and then we can compute the key $K := H(m)$. However, there is a subtlety in case of decryption failures: If $m = \bot$, then Decaps does not return $\bot$, but instead returns a key K that is indistinguishable from one that would result from a successful decryption. (This is called "implicit rejection", as opposed to "explicit rejection" that would return $\bot$.) This key K is generated from the ciphertext as $K := \mathsf{PRF}(k, c)$ where PRF is a pseudorandom function.[11] And the PRF-key k is part of the secret key of the KEM. (We cannot use $K := \tilde{H}(c)$ for some hash function $\tilde{H}$ because then the adversary could notice that K is the hash of c since $\tilde{H}$ is public.)

We describe the transformation $\mathsf{U}^{\not\perp}_m$ in Figure 4.

Security of transformation $\mathsf{U}^{\not\perp}_m$. HKSU does not show the security of transformation $\mathsf{U}^{\not\perp}_m$ (in the sense of showing that Encaps is secure if Enc' satisfies certain properties), but instead directly analyzes the result of applying both T and $\mathsf{U}^{\not\perp}_m$. That is, they show that Encaps is secure if Enc satisfies certain properties.[12] HKSU does not completely modularize the proof (i.e., it does not separately analyze T and $\mathsf{U}^{\not\perp}_m$) but shows the following:

Theorem 2 *Assume* Enc *has injective encryption*[13] *and is IND-CPA secure and DS secure and ε-disjoint, and has ε'-correctness.*[14] *(For negligible $\varepsilon, \varepsilon'$) Then* Encaps *(as constructed by transformations* T *and* $\mathsf{U}^{\not\perp}_m$ *from* Enc*) is IND-CCA secure.*

The result stated in HKSU includes concrete security bounds.

We also recall the definition of IND-CCA security for KEMs used in the preceding theorem:

[11] Note that HKSU [22] instead uses a *secret* random function H_r. (Not a public random function like the random oracle.) But it is understood that this secret random function is to be implemented by a PRF. Here, we directly use the PRF since we want to avoid keeping the proof step that replaces the PRF by a random function implicit.

[12] But note that the proof still uses the result from Theorem 1 as part of its proof, so we do have at least partially a modularized analysis.

[13] This means that for any $m_0 \neq m_1$ in the message space, $\mathsf{Enc}(pk, m_0) \neq \mathsf{Enc}(pk, m_1)$ with probability 1. Note that this does not imply the possibility of correct decryption: While information theoretically, the plaintext is determined by the ciphertext, it may be computationally infeasible to determine the correct plaintext with probability 1, even given the secret key.

[14] This means that for random $(pk, sk) \leftarrow \mathsf{Keygen}()$ and worst-case m, $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m$ with probability at least $1 - \varepsilon$. See [22] for a precise definition.

Game $\mathsf{IND\text{-}CCA}_0$
$(pk, sk) \leftarrow \mathsf{Keygen}_{\mathsf{FO}}()$
$(K^*, c^*) \leftarrow \mathsf{Encaps}(pk)$
$b \leftarrow A^{\textsc{Decaps}}(pk, c^*, K^*)$

Game $\mathsf{IND\text{-}CCA}_1$
$(pk, sk) \leftarrow \mathsf{Keygen}_{\mathsf{FO}}()$
$(K^*, c^*) \leftarrow \mathsf{Encaps}(pk)$
$K^* \xleftarrow{\$} \mathcal{K}$
$b \leftarrow A^{\textsc{Decaps}}(pk, c^*, K^*)$

Oracle $\textsc{Decaps}(c \neq c^*)$
$K \leftarrow \mathsf{Decaps}(sk, c)$
return K

Figure 5: Games in the definition of IND-CCA security. In the random oracle model, A is additionally given oracle access to all random oracles.

Definition 2 *A KEM* $(\mathsf{Keygen}_{\mathsf{FO}}, \mathsf{Encaps}, \mathsf{Decaps})$ *with key space* $\mathcal{K}$ *is IND-CCA secure iff for any quantum-polynomial-time adversary* A, $\left|\Pr[b = 1 : \mathsf{IND\text{-}CCA}_0] - \Pr[b = 1 : \mathsf{IND\text{-}CCA}_1]\right|$ *is negligible, using the games from Figure 5.*

Intuitively, this means that A cannot distinguish between the true key K^* contained in the ciphertext c^* and a uniformly random key K^*.

Note that in this definition, we slightly deviate from HKSU: In HKSU, only one game is given. This game picks randomly whether to play $\mathsf{IND\text{-}CCA}_0$ or $\mathsf{IND\text{-}CCA}_1$ from our definition. The security definition then requires that the adversary guesses which game is played with probability negligibly close to $\frac{1}{2}$. (We call this a "bit-guessing-style definition") In contrast, our definition requires the adversary to distinguish with its output bit between two games. (We call this a "distinguishing-style definition".) It is well-known that bit-guessing-style and distinguishing-style definitions are equivalent. But in the context of formal verification, it seems (according to our experiences) easier to work with distinguishing-style definitions.

Security proof of transformation $\mathsf{U}^{\not\perp}_m$. We give a compressed overview of the proof of Theorem 2 from HKSU. For details, see [22].

Fix an adversary A. By definition of IND-CCA security (Definition 2), we need to bound $|\Pr[b = 1 : \mathsf{IND\text{-}CCA}_0] - \Pr[b = 1 : \mathsf{IND\text{-}CCA}_1]|$ for the games from Figure 5.

We use essentially the same games in this proof as HKSU, with one difference: Since we decided to define IND-CCA security via a distinguishing-style definition, we need to adapt the games accordingly. All arguments from HKSU carry over trivially to our changed presentation.

The first step is to rewrite $\mathsf{IND\text{-}CCA}_0$ by unfolding the definitions of $\mathsf{Keygen}_{\mathsf{FO}}$, Encaps, Decaps. (I.e., we make all constructions introduced by T and $\mathsf{U}^{\not\perp}_m$ explicit.) In addition, we replace the PRF by a uniformly random function H_r (that is not accessible to the adversary). The resulting game is:

Game 0
$G \xleftarrow{\$} (\mathcal{M} \to \mathcal{R})$, $H_r \xleftarrow{\$} (\mathcal{C} \to \mathcal{K})$
$H \xleftarrow{\$} (\mathcal{M} \to \mathcal{K})$
$(pk, sk) \leftarrow \mathsf{Keygen}()$
$m^* \xleftarrow{\$} \mathcal{M}$
$c^* \leftarrow \mathsf{Enc}(pk, m^*; G(m^*))$
$K^* := H(m^*)$
$b \leftarrow A^{\textsc{Decaps}, H, G}(pk, c^*, K^*)$

$\textsc{Decaps}(c \neq c^*)$
$m := \mathsf{Dec}(sk, c)$
if $m = \perp$ **or** $\mathsf{Enc}(pk, m; G(m)) \neq c$
then return $K := H_r(c)$
else return $K := H(m)$

Here $(A \to B)$ denotes the set all of functions from A to B. And $\mathcal{C}$ is the ciphertext space. (Just the syntactic space, not the set of valid ciphertexts.)

From the fact that PRF is a pseudorandom function, we get that $\left|\Pr[b = 1 : \mathsf{IND\text{-}CCA}_0] - \Pr[b = 1 : \text{Game } 0]\right|$ is negligible.

Next, we chose the random oracle H differently: Instead of chosing H uniformly, we define it as the composition of a uniformly random function H_q and the encryption function $\mathsf{Enc}(pk, -; G(-))$. (The $-$ stands for the function argument.) Since Enc has injective encryption, $\mathsf{Enc}(pk, -; G(-))$

is injective, and thus H is still uniformly distributed. We get the following game (changed lines are marked with boldface line numbers):

Game 1	$\textsc{Decaps}(c \neq c^*)$
01 $G \xleftarrow{\$} (\mathcal{M} \to \mathcal{R})$, $H_r \xleftarrow{\$} (\mathcal{C} \to \mathcal{K})$	09 $m := \mathsf{Dec}(sk, c)$
02 $H_q \leftarrow (\mathcal{C} \to \mathcal{K})$	10 **if** $m = \bot$ **or** $\mathsf{Enc}(pk, m; G(m)) \neq c$
03 $H := H_q(\mathsf{Enc}(pk, -; G(-)))$	11 **then return** $K := H_r(c)$
04 $(pk, sk) \leftarrow \mathsf{Keygen}()$	**12 else return** $K := H_q(c)$
05 $m^* \xleftarrow{\$} \mathcal{M}$	
06 $c^* \leftarrow \mathsf{Enc}(pk, m^*; G(m^*))$	
07 $K^* := H_q(c^*)$	
08 $b \leftarrow A^{\textsc{Decaps},H,G}(pk, c^*, K^*)$	

Note that we additionally replaced two invocations $H(m^*)$, $H(m)$ by $H_q(c^*)$, $H_q(c)$. By construction, the new invocations return the same values. We have $\Pr[b = 1 : \text{Game } 0] = \Pr[b = 1 : \text{Game } 1]$.

In the next game hop, we change the decapsulation oracle. Instead of returning $H_r(c)$ or $H_q(c)$, depending on the result of the decryption, we now always return $H_q(c)$. The resulting game is:

Game 2	$\textsc{Decaps}(c \neq c^*)$
01 $G \xleftarrow{\$} (\mathcal{M} \to \mathcal{R})$, $H_r \xleftarrow{\$} (\mathcal{C} \to \mathcal{K})$	**09 return** $K := H_q(c)$
02 $H_q \leftarrow (\mathcal{C} \to \mathcal{K})$	
03 $H := H_q(\mathsf{Enc}(pk, -; G(-)))$	
04 $(pk, sk) \leftarrow \mathsf{Keygen}()$	
05 $m^* \xleftarrow{\$} \mathcal{M}$	
06 $c^* \leftarrow \mathsf{Enc}(pk, m^*; G(m^*))$	
07 $K^* := H_q(c^*)$	
08 $b \leftarrow A^{\textsc{Decaps},H,G}(pk, c^*, K^*)$	

Since H_r and H_q are both random functions, at the first glance it might seem that this change does not matter at all, the return value is still random. However, H_q is indirectly accessible to the adversary via H! A more careful case analysis reveals that the adversary can distinguish the two games if it finds a message m with "bad randomness". That is, a message m such that $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m; r)) \neq m$ where $r := G(m)$. If such bad randomness did not exist (i.e., when using a perfectly correct base scheme), this case would never happen. However, we do not assume perfect correctness. The solution from HKSU is to first replace the uniformly chosen $G \xleftarrow{\$} (\mathcal{M} \to \mathcal{R})$ by a G that outputs only good randomness (short: a "good G"). I.e., for each m, $G(m) := r$ is chosen uniformly from the set of all r with $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m; r)) = m$. Once we have such a good G, bad randomness does not occur any more, and we can show that switching between H_r and H_q cannot be noticed (zero distinguishing probability). And then we replace G back by $G \xleftarrow{\$} (\mathcal{M} \to \mathcal{R})$.

To show that replacing the uniform G by a good G, HKSU reduces distinguishing the two situations to distinguishing a sparse binary function F from a constant-zero function F_0 (given as an oracle). And for that distinguishing problem (called GDPB), they give a lemma that shows the impossibility of distinguishing the F and F_0.

Altogether, we get that $|\Pr[b = 1 : \text{Game } 1] - \Pr[b = 1 : \text{Game } 2]|$ is negligible.

Our formalization deviates somewhat from that proof: Instead of using the lemma about GDPB (which we would have to implement in the tool, first), we use the O2H Theorem [1] to show this indistinguishability. (We had to implement the O2H Theorem anyway because it is used in the analysis of transformation T.)

Note that this makes our bound somewhat worse. In HKSU, the proof step involving GDPB leads to a summand of $O(q^2\delta)$ in the final bound, while we achieve $O(q\sqrt{\delta})$ instead (last but one

summand of (1)). Here q is the number of queries and δ the correctness error of the underlying scheme.

In the next game, we change how the challenge ciphertext c^* is generated. Instead of encrypting m^*, we simply produce a fake ciphertext $c^* \leftarrow \overline{\mathsf{Enc}}(pk)$. The resulting game is:

Game 3

$G \xleftarrow{\$} (\mathcal{M} \to \mathcal{R})$, $H_r \xleftarrow{\$} (\mathcal{C} \to \mathcal{K})$
$H_q \leftarrow (\mathcal{C} \to \mathcal{K})$
$H := H_q(\mathsf{Enc}(pk, -; G(-)))$
$(pk, sk) \leftarrow \mathsf{Keygen}()$
$m^* \xleftarrow{\$} \mathcal{M}$
06 $c^* \leftarrow \overline{\mathsf{Enc}}(pk)$
$K^* := H_q(c^*)$
$b \leftarrow A^{\textsc{Decaps},H,G}(pk, c^*, K^*)$

$\textsc{Decaps}(c \neq c^*)$
return $K := H_q(c)$

By DS security of Enc, this fake encryption cannot be distinguished from a real encryption. (Note that the secret key is not used any more in Game 2.) Hence $|\Pr[b = 1 : \text{Game 2}] - \Pr[b = 1 : \text{Game 3}]|$ is negligible.

Finally, we change how K^* is chosen. Instead of picking $K^* := H_q(c^*)$, we chose K^* uniformly:

Game 4

$G \xleftarrow{\$} (\mathcal{M} \to \mathcal{R})$, $H_r \xleftarrow{\$} (\mathcal{C} \to \mathcal{K})$
$H_q \leftarrow (\mathcal{C} \to \mathcal{K})$
$H := H_q(\mathsf{Enc}(pk, -; G(-)))$
$(pk, sk) \leftarrow \mathsf{Keygen}()$
$m^* \xleftarrow{\$} \mathcal{M}$
$c^* \leftarrow \overline{\mathsf{Enc}}(pk)$
07 $K^* \xleftarrow{\$} \mathcal{K}$
$b \leftarrow A^{\textsc{Decaps},H,G}(pk, c^*, K^*)$

$\textsc{Decaps}(c \neq c^*)$
return $K := H_q(c)$

Since H_q is a random function, this change can only be noticed if $H_q(c^*)$ is queried somewhere else. The adversary has access to H_q via H, but through H it can only query H_q on values that are in the range of Enc. But since c^* was constructed as a fake encryption $\overline{\mathsf{Enc}}(pk)$, the ε-disjointness of $\overline{\mathsf{Enc}}$ guarantees that c^* is, with overwhelming probability, not in the range of Enc. In that case, $H_q(c^*)$ is independent from anything the adversary might query. Thus $|\Pr[b = 1 : \text{Game 3}] - \Pr[b = 1 : \text{Game 4}]|$ is negligible.

So far, we have shown that the games $\mathsf{IND\text{-}CCA}_0$ and Game 4 are indistinguishable. To show indistinguishability of $\mathsf{IND\text{-}CCA}_0$ and $\mathsf{IND\text{-}CCA}_1$, we write down a similar sequence of games Game 0' to Game 4' where K^* is chosen uniformly (as in $\mathsf{IND\text{-}CCA}_1$). We then have indistinguishability of $\mathsf{IND\text{-}CCA}_1$ and Game 4'. Game 4 and Game 4' are identical, thus $\mathsf{IND\text{-}CCA}_0$ and $\mathsf{IND\text{-}CCA}_1$ are indistinguishable, finishing the proof of IND-CCA security of Encaps.

4 Formalizing HKSU – The Specification

A proof (formal or pen-and-paper) will consist of two separate parts: A specification of the result that is proven, and the proof itself. This separation is important because if we trust that the proof is correct, we only need to read the specification. In a pen-and-paper proof, this specification will usually consist of the theorem together with all information required for interpreting the theorem, i.e., all definitions that the theorem refers to, as well as all assumptions (if they are not stated within the theorem itself). In the case of formal verification, we tend to trust the proof (because it has been verified by the computer) but we have to check the specification – does it indeed encode what we intended to prove?

```
1  lemma security_encFO:
2    abs(Pr[b=1: indcca_encFO_0(rho)] - Pr[b=1: indcca_encFO_1(rho)])
3    <=
4    abs (Pr[b=1:PRF_real (rho)] - Pr[b=1:PRF_ideal (rho)])
5    + abs (Pr[b=1:PRF_real'(rho)] - Pr[b=1:PRF_ideal'(rho)])
6    + abs (Pr[b=1:indcpa_enc0_0'''(rho)] - Pr[b=1:indcpa_enc0_1'''(rho)])
7    + 2 * sqrt(1+q) * sqrt(abs(Pr[b=1:indcpa_enc0_0''(rho)]
8                               - Pr[b=1:indcpa_enc0_1''(rho)])
9                           + 4 * q / card (msg_space()))
10   + abs (Pr[b=1:indcpa_enc0_0'(rho)] - Pr[b=1:indcpa_enc0_1'(rho)])
11   + 2 * sqrt(1+q) * sqrt(abs(Pr[b=1:indcpa_enc0_0(rho)]
12                              - Pr[b=1:indcpa_enc0_1(rho)])
13                          + 4 * q / card (msg_space()))
14   + 8 * sqrt(4 * (q+qD+2) * (q+qD+1)
15              * correctness params0 keygen0 enc0 dec0 msg_space0)
16   + 2 * correctness params0 keygen0 enc0 dec0 msg_space0.
```

Figure 6: The main theorem. File: `lemma_security_encFO.qrhl`

In this section we go through the specification part of our HKSU formalization (available at [37]). It consists roughly of five parts: The main theorem. The specification of the encryption algorithm and other functions in Isabelle/HOL. The specification of the security definitions (security games). The specification of the adversary. And the specification of the reduction-adversaries (we explain below why this is a relevant part of the specification).

The main theorem. The source code for the main theorem is shown in Figure 6. Line 2 is the IND-CCA advantage $\mathsf{Adv}_{\mathsf{CCA}}$ of the adversary attacking the KEM Encaps resulting from the transformations Punc, T, $\mathsf{U}^{\not\perp}_m$. (See Section 3.3.) $\mathsf{Adv}_{\mathsf{CCA}}$ is defined as the difference between the probability of adversary-output $b = 1$ in games `indcca_encFO_0` and `indcca_encFO_1`. We will see those games below. In line 4 we have the advantage $\mathsf{Adv}_{\mathsf{PRF}}$ of a reduction-adversary[15] against the pseudorandom function PRF, expressed as the probability-difference between games `PRF_real` and `PRF_ideal`. In line 5, we have basically the same but with respect to a different reduction-adversary. We have two reduction-adversaries for PRF since we used the pseudorandomness twice in the proof. Since the adversary is hardcoded in the games,[16] we express this in terms of further games `PRF_real'` and `PRF_ideal'`. In line 6 we have the IND-CPA advantage $\mathsf{Adv}'''_{\mathsf{CPA}}$ of a reduction-adversary against the base scheme Enc_0, expressed in terms of games `indcpa_enc0_0'''` and `indcpa_enc0_1'''`. Similarly, we have advantages $\mathsf{Adv}''_{\mathsf{CPA}}$ in lines 7–8, $\mathsf{Adv}'_{\mathsf{CPA}}$ in line 10, and $\mathsf{Adv}_{\mathsf{CPA}}$ in lines 11–12, against further reduction-adversaries. The term $\delta :=$ `correctness params0 ...` in lines 15 and 16 refers to the correctness of Enc_0, i.e., we assume Enc_0 to be δ-correct. (Cf. footnote 14 for the meaning and Figure 7 for the formalization of `correctness`.) Finally, `card (msg_space())` is the cardinality of the message space $\mathcal{M}$ of Enc. q_G, q_H, q_D are the number of queries made to the three oracles, and $q := q_G + 2q_H$. With the notation we introduced in this explanation, we can write the main theorem more readably:

$$\begin{aligned}\mathsf{Adv}_{\mathsf{CCA}} \le \mathsf{Adv}_{\mathsf{PRF}} + \mathsf{Adv}'_{\mathsf{PRF}} + \mathsf{Adv}'''_{\mathsf{CPA}} + 2\sqrt{1+q}\sqrt{\mathsf{Adv}''_{\mathsf{CPA}} + 4q/|\mathcal{M}|}\\ + \mathsf{Adv}'_{\mathsf{CPA}} + 2\sqrt{1+q}\sqrt{\mathsf{Adv}_{\mathsf{CPA}} + 4q/|\mathcal{M}|}\\ + 8\sqrt{4(q+qD+2)(q+qD+1)\delta} + 2\delta. \quad (1)\end{aligned}$$

[15] By reduction-adversary, we mean an adversary that we have explicitly constructed.

[16] Due to a lack of a proper module system in `qrhl-tool`, we have a lot of code duplication. A module system for games and adversaries such as in EasyCrypt would be a valuable addition to `qrhl-tool` and would have simplified our proofs considerably.

```
1  definition "force_into M x = (if x∈M then x else SOME m. m∈M)"
2
3  definition correctness_pkskm where "correctness_pkskm enc dec p pk sk m
4    = Prob (enc p pk m) {c. dec p sk c ≠ Some m}"
5
6  definition correctness_pksk where "correctness_pksk enc dec msg_space p pk sk
7    = (SUP m∈msg_space p. correctness_pkskm enc dec p pk sk m)"
8
9  definition correctness where "correctness P keygen enc dec msg_space =
10  expectation' P (λp. expectation' (keygen p)
11    (λ(pk,sk). correctness_pksk enc dec msg_space p pk sk))"
12
13 definition "injective_enc_pk p enc msg_space pk ⟷
14   (∀m1∈msg_space p. ∀m2∈msg_space p.
15       disjnt (supp (enc p pk m1)) (supp (enc p pk m2)))"
16
17 definition "injective_enc P keygen enc msg_space ⟷
18   (∀p∈supp P. ∀(pk,sk)∈supp (keygen p). injective_enc_pk p enc msg_space pk)"
19
20 axiomatization qD qG qH :: nat
21   where "qD ≥ 1" and "qG ≥ 1" and "qH ≥ 1"
22
23 definition "q = qG + 2 * qH"
```

Figure 7: Some definitions from `General_Definitions.thy`. See page 11 for a micro primer on Isabelle/HOL syntax.

Encryption algorithm and other definitions. In order to make sense of the main theorem, we first need to check the definitions of the KEM and the building blocks used in its construction. The simplest is the pseudorandom function PRF, defined in Figure 8, lines 1–2. The `axiomatization` command declares two constants `PRF` (the PRF) and `keygenPRF` (the key generation algorithm for the PRF, given as a distribution over keys). It furthermore axiomatizes the fact the key generation is a total distribution (axiom `keygenPRF_total`). (We do not need to axiomatize the security of `PRF`; its security is used implicitly by having $\mathsf{Adv}_{\mathsf{PRF}}$ occur in the main theorem.)

Similarly, we axiomatize the encryption scheme Enc_0 in lines 4–16. All encryption schemes in our work consist of a public parameter distribution (we only use this here for chosing the random oracles), a key generation, an encryption, a decryption algorithm, and a message space (which we allow to depend on the public parameters). The base scheme does not have public parameters, so we define `params0` as the point distribution that always returns the dummy value `()` (line 4). The key generation `keygen0` (lines 6–9) takes the public parameter and returns a distribution of public/secret key pairs. We assume that key generation is a total distribution (axiom `weight_keygen0`). Additionally we assume a function `pk_of_sk` that returns the corresponding `pk` for every `sk` in the support of `keygen`.[17] We define the encryption by first defining `enc0r`, a function that takes the public parameters, public key, message, and explicit randomness to compute a ciphertext (line 11). From this we define `enc0` as the distribution resulting from applying `enc0r` to the uniform distribution on the randomness (line 12). Decryption (`dec0`, line 13) may fail, hence the return type is `msg option`, which means it can be `None` or `Some m` with a message `m`. Finally, `msg_space0` is a non-empty set (lines 15–16). We have an additional axiom `enc0_injective` (lines 18–19) which encodes the assumption that our base scheme is injective. (Cf. footnote 13 for the meaning and Figure 7 for the formalization of `injective_enc`.)

The transformations Punc, T, and $\mathsf{U}_m^{\not\perp}$ are given in Figure 9. As with our base scheme, we always define a deterministic encryption/encapsulation that takes explicit randomness first. The

[17] This assumption is not explicit in HKSU but clearly necessary for defining the decryption in transformation T: since the decryption re-encrypts, it needs to know the public key.

```
axiomatization PRF :: "prfkey ⇒ ciph ⇒ key" and keygenPRF :: "prfkey distr"
where keygenPRF_total: "weight keygenPRF = 1"

definition "params0 = point_distr ()"

axiomatization keygen0 :: "unit ⇒ (pk * sk) distr"
           and pk_of_sk :: "sk ⇒ pk"
where pk_of_sk: "(pk,sk) ∈ supp (keygen ()) ⟹ pk_of_sk sk = pk"
  and weight_keygen0: "weight (keygen0 ()) = 1"

axiomatization enc0r :: "unit ⇒ pk ⇒ msg ⇒ rand ⇒ ciph"
definition "enc0 _ pk m = map_distr (enc0r () pk m) (uniform UNIV)"
axiomatization dec0 :: "unit ⇒ sk ⇒ ciph ⇒ msg option"

axiomatization msg_space0 :: "unit ⇒ msg set"
where nonempty_msg_space0: "msg_space0 () ≠ {}"

axiomatization where
  enc0_injective: "injective_enc params0 keygen0 enc0 msg_space0"
```

Figure 8: Building blocks: Base scheme and pseudorandom function. File: `Base_Scheme.thy` (last line: `FO_Specification.thy`). See page 11 for a micro primer on Isabelle/HOL syntax.

final KEM consists of the functions `keygenFO`, `encapsFO`, `decapsFO`, etc. We omit a discussion of the details of the function definitions, they follow our exposition in Section 3.

Security definitions / games. Next we have to understand the games that define the various advantages in the main theorem. We start with the IND-CCA security of Encaps. $\mathsf{Adv}_{\mathsf{CCA}}$ was defined as the difference in probabilities that an adversary A (`Adv_INDCCA_encFO` in our case) outputs $b = 1$ in games `indcca_encFO_0/1`. The formalization of these games is given in Figure 10. It is a direct encoding of the games in Figure 5, with several small differences: Since we do not support procedures with parameters and return values, we use the global variables `in_pk` and `in_cstar` and `Kstar` for the inputs pk and c^* and K^*. And the global variable `b` is used for the return value (guessing bit). Below, when defining the adversary, we will then make sure the adversary gets access to those global variables.[18] Access to the oracle `decapsQuery` is by passing it to the adversary as one of the oracles. Communication with `decapsQuery` is through variables `c` (input) and `K'` (output). It checks explicitly whether $c \neq c^*$ and returns `None` otherwise. (In Figure 5, it was not made explicit how we enforce $c \neq c^*$.) Additionally, we model the access to the random oracles G, H by giving A access to `queryG`, `queryH`. `queryG` operates on global variables `Gin`, `Gout` and applies the unitary transformation `Uoracle G` on them. (`Uoracle` is a built-in function that transforms a function G into a unitary $|x, y\rangle \mapsto |x, y \oplus G(x)\rangle$.) Analogously `queryH`.

Similarly we define the games used in the rhs of the main theorem. The games `PRF_real` and `PRF_ideal` defining PRF-security for adversary `Adv_PRF` are given in Figure 11. Again, we define oracles to either evaluate a pseudo-random function `PRF` or a random function `RF` and pass them to the adversary. The adversary `Adv_PRF` is explicitly defined in terms of `Adv_INDCCA_encFO` as part of our reduction, but its implementation details do not matter for us (except for some necessary sanity checks, see below). The primed variants `Adv_real'` and `Adv_ideal'` are identical except that they use a different reduction-adversary.

Similarly, we define IND-CPA security of Enc_0 against `Adv_INDCPA_enc0_1/2` in Figure 12. The primed variants are identical except that they use a different adversary.

[18] We do not use `pk` and `cstar` directly for passing pk and c^* since that would mean giving A access to those variables. Then A could change the value of pk and c^* but the oracle `decapsQuery` relies on having the original values of pk and c^*.

```
1  definition "params = params0"
2  definition "keygen = keygen0"
3  definition "encr = enc0r"
4  definition "dec P sk c = (case dec0 P sk c of Some m ⇒ if m ∈
   msg_space P then Some m else None | None ⇒ None)"
5  definition "fakeenc _ pk = enc0 () pk puncture"
6  definition "enc _ pk m = map_distr (encr () pk m) (uniform UNIV)"
7
8  definition paramsT where "paramsT = uniform UNIV"
9  definition "keygenT G = keygen ()"
10 definition "encrT G pk m = encr () pk m (G m)"
11 definition "encT G pk m = point_distr (encrT G pk m)"
12 definition "decT G sk c = (case dec () sk c of None ⇒ None
13   | Some m ⇒ if encrT G (pk_of_sk sk) m = c then Some m else None)"
14 definition "fakeencT G = fakeenc ()"
15 definition "msg_spaceT G = msg_space ()"
16
17 definition paramsFO where "paramsFO = uniform UNIV"
18 definition keyspaceFO where "keyspaceFO _ = UNIV"
19 definition "keygenFO = (λ(G,H). map_distr (λ((pk,sk),prfk). (pk,(sk,prfk)))
20              (product_distr (keygenT G) keygenPRF))"
21 definition "encapsrFO = (λ(G,H) pk r. (H(r), encrT G pk r))"
22 definition "encapsFO GH pk = map_distr (encapsrFO GH pk)
23              (uniform (msg_spaceT (fst GH)))"
24 definition "decapsFO = (λ(G,H) (sk,prfk) c. case decT G sk c of
25   None ⇒ Some (PRF prfk c) | Some m ⇒ Some(H(m)))"
```

Figure 9: Functions resulting from transformations Punc, T, $\mathsf{U}_m^{\not\perp}$. Files: `Punc_Specification.thy` (l. 1–6), `T_Specification.thy` (l. 8–15), `FO_Specification.thy` (l. 17–25). See page 11 for a micro primer on Isabelle/HOL syntax.

The adversary. In the games `indcca_encFO_1/2`, we use the adversary $A :=$ `Adv_INDCCA_encFO`. Since we want the main theorem to hold for arbitrary adversaries, we need to declare the adversary as an unspecified program. This is done in Figure 13. It declares that the adversary has access to the variables `classA`, `quantA`, `b`, `in_pk`, `in_cstar`, `Kstar`, i.e., we say the adversary has those free variables. Here `classA`, `quantA` are the global state of the adversary (consisting of one quantum and one classical variable), and the others are the variables used for inputs/outputs of the adversary. Furthermore, the adversary needs to be able to access the variables `Hin`, `Hout`, `Gin`, `Gout`, `c`, `K'` that are used as inputs/outputs for its oracles `decapsQuery`, `queryG`, `queryH` (see above). Those variables are not declared as free variables (i.e., the adversary will have to hide them under a `local` command) but may be used internally, in particular before or after invoking the oracle. Finally, `calls ?,?,?` means that the adversary takes three oracles.

However, we are not interested in arbitrary adversaries, but in ones that always terminate and that make $\leq q_G, q_H, q_D$ queries to its various oracles. For this, we add various axioms to the file `axioms.qrhl`, stating the termination and the number of queries performed when instantiated with various oracles. The file is reproduced in Appendix A. Unfortunately, this file contains a lot of repetitions because `qrhl-tool` does not allow us to allquantify over the oracles, so we need to state the axioms for any oracle we want to instantiate the adversary with.[19]

Reduction-adversaries. Finally, to fully check whether the main theorem states what we want it to state (namely, that the KEM Encaps is secure assuming that the underlying encryption scheme Enc_0 and the PRF are secure), we also need to inspect the reduction-adversaries. This is because the main theorem basically says: If `Adv_INDCCA_encFO` breaks Encaps, then one of

[19] Another place where a more advanced module system would help, cf. footnote 16.

```
program indcca_encFO_0 := {
  (G,H) <$ paramsFO;
  (pk,skfo) <$ keygenFO (G,H);
  (Kstar,cstar) <$ encapsFO (G,H) pk;
  in_pk <- pk;
  in_cstar <- cstar;
  call Adv_INDCCA_encFO
    (queryG,queryH,decapsQuery);
}.

program indcca_encFO_1 := {
  (G,H) <$ paramsFO;
  (pk,skfo) <$ keygenFO (G,H);
  (Kstar,cstar) <$ encapsFO (G,H) pk;
  Kstar <$ uniform (keyspaceFO (G,H));
  in_pk <- pk;
  in_cstar <- cstar;
  call Adv_INDCCA_encFO
    (queryG,queryH,decapsQuery);
}.

program queryG := {
  on Gin,Gout apply (Uoracle G);
}.

program queryH := {
  on Hin,Hout apply (Uoracle H);
}.

program decapsQuery := {
  if (c=cstar) then
    K' <- None;
  else
    K' <- decapsFO (G,H) skfo c;
}.
```

Figure 10: IND-CCA security definition for Encaps. Files: `indcca_encfo_0.qrhl`, `indcca_encfo_1.qrhl`, `decapsQuery.qrhl`, `queryG.qrhl`, `queryH.qrhl`.

```
program PRF_real := {
  prfk <$ keygenPRF;
  call Adv_PRF(queryPRF);
}.

program PRF_ideal := {
  RF <$ uniform UNIV;
  call Adv_PRF(queryRF);
}.

program queryPRF := {
  K <- PRF prfk c;
}.

program queryRF := {
  K <- RF c;
}.
```

Figure 11: Pseudorandomness game for `Adv_PRF`. Files `PRF_real.qrhl`, `PRF_ideal.qrhl`, `queryPRF.qrhl`, `queryRF.qrhl`.

the adversaries in the games on the rhs breaks Enc_0 or PRF. (I.e., one of `Adv_PRF`, `Adv_PRF'`, `Adv_INDCPA_enc0/1`, etc.) But this is vacuously true – it is easy to construct an adversary that breaks Enc_0 or PRF. Namely, that adversary could run in exponential-time and perform a brute-force attack. Or that adversary could directly access the global variables containing, e.g., the secret key. So, while the exact details of what the reduction-adversaries do are not important, we need to check: Are the reduction-adversaries quantum-polynomial-time if `Adv_INDCCA_encFO` is? (Or even some more refined runtime relationship if we want tight concrete security bounds.) And do the reduction-adversaries access only variables that are not used by the security games themselves? The latter can be checked using the `print` command in interactive mode that prints all variables of a program (e.g., `print Adv_PRF`). This shows that the adversaries in the PRF games only access `cstar`, `classA`, `b`, `c`, `K'`, `quantA`, and in particular not `prfk` or `RF`. And the adversaries in the IND-CPA games access only `Find`, `mstar`, `S`, `in_cstar`, `in_pk`, `classA`, `b`, `is_puncture`, `G`, `quantA`, but not the forbidden `sk`, `pk`, `cstar`.[20] To check the runtime of the adversaries, there is currently no better way than to manually inspect the code of all adversaries explicitly to see whether they do anything that increases the runtime too much.[21] To the best

[20] Again, a more refined module system would allow us to automatically derive that certain variable-disjointness conditions hold, cf. footnote 16.

[21] Including recursively invoked programs, we need to check: `Adv_PRF`, `Adv_PRF'`, `Adv_INDCPA_enc0_1`, `Adv_INDCPA_enc0_2`, `Adv_INDCPA_enc0_1'`, `Adv_INDCPA_enc0_2'`, `Adv_INDCPA_enc0_2''`,

```
program indcpa_enc0_0 := {
  (pk,sk) <$ keygen0 ();
  in_pk <- pk;
  call Adv_INDCPA_enc0_1;
  m0star <- force_into
            (msg_space0()) m0star;
  m1star <- force_into
            (msg_space0()) m1star;
  cstar <$ enc0 () pk m0star;
  in_pk <- pk;
  in_cstar <- cstar;
  call Adv_INDCPA_enc0_2;
}.
```

```
program indcpa_enc0_1 := {
  (pk,sk) <$ keygen0 ();
  in_pk <- pk;
  call Adv_INDCPA_enc0_1;
  m0star <- force_into
            (msg_space0()) m0star;
  m1star <- force_into
            (msg_space0()) m1star;
  cstar <$ enc0 () pk m1star;
  in_pk <- pk;
  in_cstar <- cstar;
  call Adv_INDCPA_enc0_2;
}.
```

Figure 12: IND-CPA security definition of Enc_0 for `Adv_INDCPA_enc0_1/2`. Files `indcpa_enc0_1.qrhl`, `indcpa_enc0_0.qrhl`.

```
adversary Adv_INDCCA_encFO
      vars classA, quantA, b, in_pk, in_cstar, Kstar
      inner Hin, Hout, Gin, Gout, c, K'  calls ?,?,?.
```

Figure 13: Adversary declaration. File: `Adv_INDCCA_encFO.qrhl`.

of our knowledge, this is the state-of-the-art also in classical crypto verification. We believe that coming up with formal verification support for runtime analysis in `qrhl-tool` and similar tools is a very important next step. If this would be solved, the reduction-adversaries could be removed from the list of things we need to check as part of the specification.

By checking all the above points, we can have confidence that the formal proof indeed proves the right thing. (There are quite a lot of points to check, but we stress that in a pen-and-paper proof, the situation is similar – one needs to check whether all security definitions are correct, etc.)

5 Formalizing HKSU – The Proof

Since the formal proof is much too long to go through in detail, we only show a few select elements here to given an impression. HKSU shows security of three transformations Punc, T, $\mathsf{U}^{\not\perp}_m$. The proof follows the overall structure of HKSU, `lemma_ds_security.qrhl` and `lemma_indcpa_security.qrhl` establishing DS and IND-CPA security of Punc, `lemma_ds_encT_security.qrhl` establishing DS security of T, and `lemma_encFO_indcca.qrhl` establishing IND-CCA security of the combination of T and $\mathsf{U}^{\not\perp}_m$. Finally `lemma_security_encFO.qrhl` combines all those results into one overall result, the "main theorem" discussed in Section 4.

`lemma_encFO_indcca.qrhl` establishes IND-CCA security using the same sequence of games as described in Section 3.3, encoded as programs `game0FO`, ..., `game4FO`, `game3FO'`, ..., `game0FO'` in the eponymous files. (The primed variants are very similar to the nonprimed ones and are autogenerated by search & replace.)

`Adv_INDCPA_enc0_2'''`, `decapsQueryPRF`, `Adv_INDCPA_enc_1`, `Adv_INDCPA_enc_2`, `Adv_INDCPA_enc_2'`, `Adv_DS_enc`, `Adv_DS_enc'`, `queryGPuncturedS`, `Adv_DS_encT`, `Adv_DS_encT'`, `queryH_Hq`, `decapsQuery2_adv_cstar`. This is a lot to check but the checks are fortunately very simple.

Game 1 to Game 2. We zoom in some more onto the proof of the relationship between Game 1 and Game 2 (`lemma_game1FO_game2FO.qrhl`). We follow the basic intuition from Section 3.3, and split the proof of that step into the following subgames (all in eponymous `.qrhl` files):

(1) `game1FO`: Game 1 from Section 3.3.

(2) `game1FO_goodbad`: In this game, we prepare for replacing uniform G by a good G. For this purpose, instead of picking G uniformly, we pick a good `Ggood` (i.e., picking `Ggood`(m) uniformly from the good randomnesses for every m) and a bad `Gbad`, and a set `S` of messages. We define $G(m)$ to be `Ggood`(m) if $m \notin$ `S` and `Gbad`(m) otherwise. By choosing the distribution of `S` properly, we have that the resulting G is still uniform.
We additionally remove all direct access to G, and make sure that `queryG` is used everywhere instead. This is necessary for bringing the game into the shape needed in the following step. This means all classical queries to G (e.g., in the creation of the challenge ciphertext) need to be replaced by quantum queries with subsequent measurements (we define a wrapper oracle `ClassicalQueryG(queryG)` for this), and we cannot simply define the function H in terms of G (see Game 1, line 03 in Section 3.3). Instead, we need to construct an oracle `queryH_Hq` that implements superposition queries of H in terms of superposition queries of G (via `queryG`). This makes this proof step considerably more complex than many of the other game steps.
That $\Pr[b = 1]$ does not change is shown in `lemma_game1FO_goodbad.qrhl`.

(3) `game1FO_goodbad_o2h_right`: We rewrite the previous game to have the right shape for the O2H theorem. The O2H theorem allows us to replace one oracle by another one that differs only in a few (hard to find) places. In order to apply the O2H theorem [1] (or the `o2h` tactic in `qrhl-tool`), the game needs to have a very specific form: `count` $\leftarrow 0; (S, G, G', z') \xleftarrow{\$} \mathcal{D};$ { **local** V; `call` A_{O2H}(`Count`(`query`)) } for an oracle `Count` that counts queries in variable `count` and `query` that implements superposition queries to G'. The distribution $\mathcal{D}$ and the program A_{O2H} can be chosen freely. In our case we choose $\mathcal{D} :=$ `goodbad_o2h_distr` such that G' is G from the previous game, and G is `Ggood`, and we choose $A_{O2H} =$ `Adv_O2H_Game1FO` to simulate the rest of the game. We show that the probability of $\Pr[b = 1]$ does not change (`lemma_game1FO_goodbad_o2h_right.qrhl`).

(4) `game1FO_goodbad_o2h_left`: We replace queries to G' by queries to G (recall that G was, in the previous game, made to return only good randomness). The Semiclassical O2H theorem [1] (implemented via our tactic `o2h`) allows us to do this replacement. In the resulting game $\Pr[b = 1]$ will differ by an amount that can be bounded in terms of the probability of finding an element in `S`. Bounding this probability involves a side-chain of games that we omit here. Altogether, `lemma_game1FO_o2h_concrete.qrhl` gives a concrete bound on the difference of $\Pr[b = 1]$.

(5) `game1FO_goodbad_o2h_left'`: We remove the query-counting wrapper oracle `Count` that was introduced for the `o2h` tactic. We do this in a separate game step because it would be in the way in the next step. The probability $\Pr[b = 1]$ does not change (`lemma_game1FO_goodbad_o2h_left'.qrhl`).

(6) `game1FO_goodbad_o2h_left_class`: We unwrap the adversary `Adv_O2H_Game1FO` again which we introduced in (3). We also undo the various replacements done in (2) (which ensured that G was never used directly) to make the game simpler for the following steps. The probability $\Pr[b = 1]$ does not change (`lemma_game1FO_goodbad_o2h_left_class.qrhl`).

(7) `game1FO_goodbad_badpk`: In (2), we ignored one problem: Even if there is just one m without any good randomness, then it is not well-defined to pick G uniformly from the set of good G's because that set is empty.[22] For that reason, in (2), we actually defined $G(m)$ to be good *if good randomness exists*. But this definition breaks the next step below which relies on the fact that all randomness is good. Our solution is to introduce a predicate `bad_pk pk sk` that tells us whether there is an m (for that key pair) without

[22] This problem also exists in HKSU but was not noticed there.

good randomness. We then change the definition of the game to make a case distinction on `bad_pk pk sk`. If true, the new game behaves in a way that makes the next proof step trivially true. If false, the new game behaves as before. The probability for `bad_pk pk sk` is bounded by the correctness error of Enc_0, so we can bound the difference of $\Pr[b = 1]$ in `lemma_game1FO_goodbad_badpk.qrhl`.

(8) `game2FO_goodbad_range`: In the previous games, the choice whether DECAPS returns $H_r(c)$ or $H_q(c)$ depended on whether we have a reencryption failure or not. (See DECAPS in Game 1 in Section 3.3.) Instead, we use $H_r(c)$ or $H_q(c)$ depending on whether c is in the range of Enc'. We can show that, assuming good randomness, these two conditions are equivalent. Since G contains only good randomness, $\Pr[b = 1]$ does not change (`lemma_game1FO_game2FO_o2h.qrhl`).

(9) `game2FO_goodbad_o2h_left'`: In the previous game, Decaps returns $H_r(c)$ if c is not in the range of Enc'. We replace this by always returning $H_q(c)$ as in Game 2 (Section 3.3). By analysis of the game, we can see that H_q is used in other places of the game only on the range of $\mathsf{Enc}' =$ `encT`, hence $H_q(c)$ and $H_r(c)$ are both fresh randomness if c is not in the range. Hence the replacement does not change $\Pr[b = 1]$ (`lemma_game2FO_goodbad_range.qrhl`).

(10) The rest of the proof steps are analogous to those done in (2)–(6), in reverse order (`lemma_game2FO_goodbad_o2h_left_class.qrhl`, `lemma_game2FO_goodbad_o2h_left'.qrhl`, `lemma_game2FO_o2h_concrete.qrhl`, `lemma_game2FO_goodbad_o2h_right.qrhl`, `lemma_game2FO_goodbad.qrhl`) until we reach `game2FO`.

Verification of `ClassicalQueryG`. To finish our illustration, we give the details of one of the subproofs of step (2), namely the proof that accessing G directly is the same querying G via `ClassicalQueryG(queryG)`. The source of `ClassicalQueryG` is given in Figure 14, lines 1–6. It initializes `Gin` with $|$`gin`$\rangle$, `Gout` with $|$`gout`$\rangle$, calls the query oracle (which will query G in superposition), and measures `Gout` into `gout`. Lines 8–11 claim that after doing so (in the right program) we will have `gout2 = G2(gin2)`. And furthermore, that this preserves quantum equality of `quantA`, `aux` between the left and right side. Lines 13–14 inlines the definitions of the programs that we use, and lines 16–17 removes the local variable declarations. (The subgoal now has the same pre-/postcondition as before, but the right program is the code of `ClassicalQueryG` without the `local` statement.) Then `wp right` (line 19) consumes the statement `gout <- measure Gout with computational_basis`, and the postcondition becomes (after simplification) what is written in lines 20–21. Basically, this proof step tells us that having $|$`gin2`$, $`G2 gin2`$\rangle$ in `Gin2,Gout2` is sufficient for having `gout2 = G2(gin2)` after measurement. Next (lines 24–27) we consume "`on Gin,Gout apply (Uoracle G)`" from `queryG` (see Figure 10, evaluation of G in superposition) and show that now it is sufficient to have $|$`gin2`$, 0\rangle$ in `Gin2,Gout2`. In lines 29–33, we remove the initialization `Gout <q ket 0`, now the necessary condition is to have $|$`gin`$\rangle$ in `Gin2`. And in lines 35–37, we remove `Gin <q ket gin`, removing the last requirement. Now left and right program are both `skip` and the pre-/postcondition are identical. The `skip` tactic (line 39) solves such a qRHL subgoal.

```
1  program ClassicalQueryG(query) := {
2    local Gin, Gout;
3    Gin <q ket gin;
4    Gout <q ket 0;
5    call query;
6    gout <- measure Gout with computational_basis; }.
7
8  qrhl ClassicalQueryG_queryG:
9   {top ⊓ ⟦quantA1,aux1⟧ ≡q ⟦quantA2,aux2⟧}
10    skip; ~ call ClassicalQueryG(queryG);
11  {Cla[gout2 = G2(gin2)] ⊓ ⟦quantA1,aux1⟧ ≡q ⟦quantA2,aux2⟧}.
12
13  inline ClassicalQueryG.
14  inline queryG.
15
16  local remove right.
17   simp!.
18
19  wp right.
20  conseq post: ⟦quantA1,aux1⟧ ≡q ⟦quantA2,aux2⟧
21               ⊓ Span {ket (gin2, G2 gin2)}»⟦Gin2, Gout2⟧.
22   simp! aux5a aux5b.
23
24  wp right.
25  conseq post: ⟦quantA1,aux1⟧ ≡q ⟦quantA2,aux2⟧
26               ⊓ Span {ket (gin2, 0)}»⟦Gin2, Gout2⟧.
27   simp! applyOpSpace_Span.
28
29  wp right.
30  conseq post: ⟦quantA1,aux1⟧ ≡q ⟦quantA2,aux2⟧
31               ⊓ Span {ket gin2}»⟦Gin2⟧.
32  rule aux6.
33   simp!.
34
35  wp right.
36  conseq post: ⟦quantA1,aux1⟧ ≡q ⟦quantA2,aux2⟧.
37   simp! leq_space_div.
38
39  skip.
40  simp!.
41 qed.
```

Figure 14: Verification of `ClassicalQueryG`. Files: `ClassicalQueryG.qrhl` (l. 1–6), `lemma_ClassicalQueryG_queryG.qrhl` (l. 8–41).

A Axioms about the adversary

The following lists all the axioms about the adversary `Adv_INDCCA_encFO` that we assume. The can be found in file `axioms.qrhl`. The axioms with postcondition `Cla[True]` axiomatize that the adversary terminates in various conditions, the others axiomatize the number of queries the adversary makes.

The fact that a program A terminates can, by semantics of qRHL, compactly be expressed as $\{\mathfrak{Cla}[\texttt{true}]\}\, A \sim \textbf{skip}\, \{\mathfrak{Cla}[\texttt{true}]\}$. By definition, this means for any initial states, A and the empty program **skip** terminate with the same probability, and their final states satisfy the predicate $\mathfrak{Cla}[\texttt{true}]$. Since $\mathfrak{Cla}[\texttt{true}]$ is always satisfied, and since **skip** terminates with probability 1, this means that A terminates with probability 1 (and nothing more than that).

Since $A :=$ Adv_INDCCA_encFO takes three oracles, we would like to express something like "for all oracles $\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3$ (that are themselves terminating), $\{\mathfrak{Cla}[\texttt{true}]\}\, A(\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3) \sim \textbf{skip}\, \{\mathfrak{Cla}[\texttt{true}]\}$". Unfortunately, `qrhl-tool` does not support quantification over programs, hence we instead give a number of axioms of this form, for the different oracles $\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3$ that occur throughout our proof. Each of those statements is "proven" with the tactic `admit` which axiomatizes the proof goal.

```
qrhl Adv_INDCCA_encFO_lossless_dq2: {Cla[True]}
     call Adv_INDCCA_encFO(queryG,queryH,decapsQuery2); ~ skip; {Cla[True]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_lossless_dq1: {Cla[True]}
     call Adv_INDCCA_encFO(queryG,queryH,decapsQuery1); ~ skip; {Cla[True]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_lossless_bpk: {Cla[True]}
     call Adv_INDCCA_encFO(queryG,queryH,decapsQuery1_badpk); ~ skip; {Cla[True]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_lossless_Gmeas_Hq_ac: {Cla[True]}
     call Adv_INDCCA_encFO(Count(queryGmeasured),queryH_Hq(Count(queryGmeasured)),
          decapsQuery2_adv_cstar); ~ skip; {Cla[True]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_lossless_C_Gmeas_Hq_dq1G: {Cla[True]}
     call Adv_INDCCA_encFO(Count(queryGmeasured),queryH_Hq(Count(queryGmeasured)),
          decapsQuery1_G(Count(queryGmeasured))); ~ skip; {Cla[True]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_lossless_C_Gmeas_Hq_dq2G: {Cla[True]}
     call Adv_INDCCA_encFO(Count(queryGmeasured),queryH_Hq(Count(queryGmeasured)),
          decapsQuery2_G(Count(queryGmeasured))); ~ skip; {Cla[True]}.
 admit.
qed.
```

Furthermore, we need to axiomatize the fact that $A :=$ Adv_INDCCA_encFO calls its first oracle $\leq q_G$ times, its second oracle $\leq q_H$ times, and its third oracle $\leq q_D$ times. As with the termination, this property cannot be stated generally, instead we encode it for different combinations of oracles given to A separately, and for each such case, explicitly count the queries to one of the oracle of interest. For example, the first case is that we give the oracles `queryG`, `queryH_Hq(queryG)` and `decapsQuery1` to A, and want to specify that this implies $\leq q_G + 2q_H$ queries to `queryG`. (There is a factor of 2 in there because `queryH_Hq`, see Figure 15, invokes `queryG` twice.) To count this, we define a wrapper oracle `Count(X)` (Figure 15) that, when invoked, increases a

```
program queryH_Hq(queryGG) := {
  local Gin, Gout;
  Gout <q ket 0;

  on Hin,Gin apply comm_op;
  call queryGG;

  on (Gin,Gout),Hout apply Uoracle (%(m,g). mk_Hq' Hq H0 g pk m);

  call queryGG;
  on Hin,Gin apply comm_op;
}.

program Count(O) := {
  call O;
  count <- count + 1;
}.
```

Figure 15: Oracle `queryH_Hq` and wrapper `Count`.

counter `count` and runs `X`. Thus, to state that $A(\texttt{queryG}, \texttt{queryH\_Hq}(\texttt{queryG}), \texttt{decapsQuery1})$ calls `queryG` at $\leq q_G + 2q_H$ times, we state that if initially `count` $= 0$, after running $A(\texttt{Count}(\texttt{queryG}), \texttt{queryH\_Hq}(\texttt{Count}(\texttt{queryG})), \texttt{decapsQuery1})$, `count` $\leq q_G + 2q_H$. This is expressed by the following axiomatized qRHL judgment. All following judgments follow the same ideas, for slightly different combinations of oracles.

```
qrhl Adv_INDCCA_encFO_countG: {Cla[count1=0]}
     call Adv_INDCCA_encFO(Count(queryG),queryH_Hq(Count(queryG)),decapsQuery1);
     ~ skip; {Cla[count1 ≤ qG+2*qH]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_countG_Hq_ac: {Cla[count1=0]}
     call Adv_INDCCA_encFO(Count(queryG),queryH_Hq(Count(queryG)),
          decapsQuery2_adv_cstar);~ skip;
     {Cla[count1 ≤ qG+2*qH]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_countG'_Hq_ac: {Cla[count1=0]}
     call Adv_INDCCA_encFO(Count(queryG'),queryH_Hq(Count(queryG')),
          decapsQuery2_adv_cstar); ~ skip;
     {Cla[count1 ≤ qG+2*qH]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_count_decapsQuery1_G: {Cla[count1=1]}
     call Adv_INDCCA_encFO(Count(queryG),queryH_Hq(Count(queryG)),
          decapsQuery1_G(Count(queryG))); ~ skip;
     {Cla[count1 ≤ qG+2*qH+1+qD]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_count_decapsQuery2_G: {Cla[count1=1]}
     call Adv_INDCCA_encFO(Count(queryG),queryH_Hq(Count(queryG)),
          decapsQuery2_G(Count(queryG))); ~ skip;
     {Cla[count1 ≤ qG+2*qH+1+qD]}.
 admit.
qed.

qrhl Adv_INDCCA_encFO_countG': {Cla[count1=0]}
```

```
36      call Adv_INDCCA_encFO(Count(queryG'),queryH_Hq(Count(queryG')),decapsQuery1);
37      ~ skip; {Cla[count1 ≤ qG+2*qH]}.
38  admit.
39 qed.
40
41 qrhl Adv_INDCCA_encFO_count_decapsQuery1_G': {Cla[count1=1]}
42      call Adv_INDCCA_encFO(Count(queryG'),queryH_Hq(Count(queryG')),
43           decapsQuery1_G(Count(queryG'))); ~ skip;
44      {Cla[count1 ≤ qG+2*qH+qD+1]}.
45  admit.
46 qed.
47
48 qrhl Adv_INDCCA_encFO_count_decapsQuery2_G': {Cla[count1=1]}
49      call Adv_INDCCA_encFO(Count(queryG'),queryH_Hq(Count(queryG')),
50           decapsQuery2_G(Count(queryG'))); ~ skip;
51      {Cla[count1 ≤ qG+2*qH+qD+1]}.
52  admit.
53 qed.
54
55 qrhl Adv_INDCCA_encFO_countGpunc: {Cla[count1=0]}
56      call Adv_INDCCA_encFO(Count(queryGPuncturedS),
57      queryH_Hq(Count(queryGPuncturedS)),decapsQuery1); ~ skip;
58    {Cla[count1 ≤ qG+2*qH]}.
59  admit.
60 qed.
61
62 qrhl Adv_INDCCA_encFO_countGpunc_Hq_ac: {Cla[count1=0]}
63      call Adv_INDCCA_encFO(Count(queryGPuncturedS),
64           queryH_Hq(Count(queryGPuncturedS)),decapsQuery2_adv_cstar);
65      ~ skip; {Cla[count1 ≤ qG+2*qH]}.
66  admit.
67 qed.
68
69 qrhl Adv_INDCCA_encFO_countGmeas_Hq_ac: {Cla[count1=0]}
70      call Adv_INDCCA_encFO(Count(queryGmeasured),queryH_Hq(Count(queryGmeasured)),
71           decapsQuery2_adv_cstar); ~ skip;
72      {Cla[count1 ≤ qG+2*qH]}.
73  admit.
74 qed.
75
76 qrhl Adv_INDCCA_encFO_count_Gmeas_Hq_dq1G_1: {Cla[count1=1]}
77      call Adv_INDCCA_encFO(Count(queryGmeasured),queryH_Hq(Count(queryGmeasured)),
78           decapsQuery1_G(Count(queryGmeasured)));
79      ~ skip; {Cla[count1 ≤ qG+2*qH+qD+1]}.
80  admit.
81 qed.
82
83 qrhl Adv_INDCCA_encFO_count_Gmeas_Hq_dq2G_1: {Cla[count1=1]}
84      call Adv_INDCCA_encFO(Count(queryGmeasured),queryH_Hq(Count(queryGmeasured)),
85           decapsQuery2_G(Count(queryGmeasured)));
86      ~ skip; {Cla[count1 ≤ qG+2*qH+qD+1]}.
87  admit.
88 qed.
89
90 qrhl Adv_INDCCA_encFO_count_decapsQuery1_Gpunc: {Cla[count1=1]}
91      call Adv_INDCCA_encFO(Count(queryGPuncturedS),queryH_Hq(Count(queryGPuncturedS)),
92           decapsQuery1_G(Count(queryGPuncturedS)));
93      ~ skip; {Cla[count1 ≤ qG+2*qH+qD+1]}.
94  admit.
95 qed.
96
97 qrhl Adv_INDCCA_encFO_count_decapsQuery2_Gpunc: {Cla[count1=1]}
98      call Adv_INDCCA_encFO(Count(queryGPuncturedS),queryH_Hq(Count(queryGPuncturedS)),
```

```
99              decapsQuery2_G(Count(queryGPuncturedS)));
100       ~ skip; {Cla[count1 ≤ qG+2*qH+qD+1]}.
101   admit.
102 qed.
```

B O2H Theorem

As part of our formalization of HKSU, we implemented support for using the O2H Theorem [1] in `qrhl-tool`. In this section, we give a short recap of the O2H Theorem and describe the new tactic.

B.1 The theorems

We state one case of the O2H Theorem from [1]. (The original theorem additionally contains several alternative formulations. It also supports tighter bounds for the case that the adversary performs multiple oracle queries in parallel. The form we state here is the main variant of the theorem.)

Theorem 3 (Semi-classical O2H [1, Thm. 1]) *Let $S \subseteq X$ be random. Let $G, H : X \to Y$ be random functions satisfying $\forall x \notin S.\ G(x) = H(x)$. Let z be a random bitstring. (S, G, H, z may have arbitrary joint distribution.)*

Let A be an oracle algorithm making $\leq q$ queries.

Let

$$P_{\text{left}} := \Pr[b = 1 : b \leftarrow A^H(z)]$$
$$P_{\text{right}} := \Pr[b = 1 : b \leftarrow A^G(z)]$$
$$P_{\text{find}} := \Pr[\mathsf{Found} : A^{G\setminus S}(z)]$$

Then

$$|P_{\text{left}} - P_{\text{right}}| \leq 2\sqrt{(q+1) \cdot P_{\text{find}}}$$

In this theorem, the notation $G \setminus S$ represents a "punctured" oracle: Whenever this oracle is queried, instead of applying the unitary $U_G : |x, y\rangle \mapsto |x, y \oplus G(x)\rangle$, it first measures whether the x-register contains a value in S (i.e., it performs a measurement with projector $P_S := \sum_{x\in S} |x\rangle\langle x|$). If the measurement succeeds, we say the event Found occurs. And only then $G \setminus S$ applies the unitary U_S. (In other words, besides applying G, we measure whether we found one of the marked values in S.)

What is this theorem good for?

Consider two games that are identical, except that the random oracle[23] used in the two games is allowed to differ in a small number of values. (For example, the first game might use a random function G, while the second game we use the function H which equals G except at one position x, where we set $H(x)$ to be some challenge value.) Then we can use the O2H Theorem to bound the probability with which an adversary A distinguishes the two games in terms of the (usually easier to bound) probability that A finds an input where G and H differ.

The two games to be distinguished can always be rewritten as in $P_{\text{left}}, P_{right}$ in the theorem (by using a suitable distribution of S, G, H, z), where S is the set of places where the oracles may differ (in our example, $S = \{x\}$) and z is arbitrary auxiliary information. Then the game where the adversary tries to find an input where G and H differ is given as in P_{find}. The event Found is raised by the oracle $G \setminus S$ when the adversary queries a value in S. Thus, we are now left to analyze a search problem instead of a distinguishing problem.

[23] We say random oracle here for simplicity, although the O2H Theorem does not require that this oracle is uniformly distributed.

There is one caveat, though: The punctured oracle $G \setminus S$ performs a partial measurement of the input register. This makes the analysis of the probability that A finds a value in S harder. For example, it is possible that the measurement whether the input is in S leaks information about S to A (even if the measurement outcome is not given to the adversary) and thus help A to find an $x \in S$. Fortunately, [1] provides the following theorem to deal with this:[24]

Theorem 4 (Search in semi-classical oracle [1, Thm. 2]) *Let A be any quantum oracle algorithm making some number of queries at depth at most d to a semi-classical oracle with domain X. Let $S \subseteq X$ and z be of arbitrary type. (S, z may have arbitrary joint distribution.)*

Let B^G be an algorithm that on input z chooses $i \xleftarrow{\$} \{1, \dots, d\}$; runs $A^G(z)$ until (just before) the i-th query; then measures all query input registers in the computational basis and outputs the measurement outcome.

Then

$$\Pr[\mathsf{Found} : A^{G \setminus S}(z)] \leq 4d \cdot \Pr[\mathit{guess} \in S : \mathit{guess} \leftarrow B^G(z)] \tag{2}$$

So this theorem allows us to upper bound the probability of finding a value in the punctured oracle $G \setminus S$ (such as in the game P_{find} above) by the probability that a related adversary (that runs A and measures a random query) finds an element of S. But B^G uses the regular oracle G (with no measurement involved). Thus the rhs (which we now need to bound) is a game in the regular setting (where oracles do not partially measure their inputs). (How to actually bound the rhs is now a different question that depends on the specific setting. For example, it might be that G, z and $S = \{x\}$ are independent, and then the rhs is bounded by the probability of guessing a uniformly random x.)

B.2 Tactics

We implemented two tactics in `qrhl-tool`, one for invocations of the O2H Theorem (Theorem 3), and one for invocations of Theorem 4.

O2H Tactic `o2h`. To apply the O2H Theorem, we have the tactic `o2h`. As a precondition for applying this tactic, the games listed in Figure 16 must be defined. The games must be defined exactly as written there, except that the names of the games, as well as the names of the variables (IN, OUT, G, S, H, z, in_S, found, count) may be chosen arbitrarily. And `distr` can be an arbitrary constant expression (meaning, the expression must not contain any program variables but may contain ambient variables). Furthermore, we require that the type of the oracle outputs (i.e., β if G has type $\alpha \Rightarrow \beta$) is of type class `xor_group`, otherwise `Uoracle` is not well-defined.

That is, `queryG` and `queryH` are implementations of the oracles that perform superposition queries to the functions G and H (using input/output registers IN, OUT). `Count` is a wrapper oracle that counts oracle queries (to express the bound on the number of queries performed by A). Let the programs `left`, `right` are just the programs defined in $P_{\mathit{left}}, P_{\mathit{right}}$ in the O2H Theorem. (Except that we additionally added a counter `count` that explicitly counts the oracle queries.) Finally, `queryGS` implements the punctured oracle $G \setminus S$ and stores in the variable `found` whether a value in S was queried. (In the definition of that program, "`proj_classical_set` S" is the projector P_S, and `binary_measurement` constructs a binary measurement from that projector.) Thus the game `find` corresponds to P_{find}.

Since the games have to be in this precise form, the first step before applying the tactic `o2h` will be to rewrite the games of interest in this specific form (for a suitably defined distribution `distr`) and show that the original and rewritten game have the same probability of $b = 1$.

[24] Again, we give the special case that does not improve the bound for adversaries making parallel queries. Furthermore, we slightly change the theorem: In our presentation, A gets the oracle G or $G \setminus S$, respectively. In the original, A gets the oracle $\mathcal{O}^{SC}_{\varnothing}$ (that does nothing) or $\mathcal{O}^{SC}_{S}$ (that only measures whether the input register is in S), respectively. Our form of the theorem is an immediate consequence of the original by using the fact that G is the same as applying $\mathcal{O}^{SC}_{\varnothing}$ and then G, and $G \setminus S$ is the same as applying $\mathcal{O}^{SC}_{S}$ and then G.

```
program queryG := {
  on IN, OUT apply (Uoracle G);
}.

program queryGS := {
  in_S <- measure IN with binary_measurement (proj_classical_set S);
  if (in_S=1) then found <- True; else skip;
  call queryG;
}.

program queryH := {
  on IN, OUT apply (Uoracle H);
}.

program Count(O) := {
  call O;
  count <- count + 1;
}.

program left := {
  count <- 0;
  (S,G,H,z) <$ distr;
  { local vars; call A(Count(queryG)); }
}.

program right := {
  count <- 0;
  (S,G,H,z) <$ distr;
  { local vars; call A(Count(queryH)); }
}.

program find := {
  count <- 0;
  (S,G,H,z) <$ distr;
  found <- False;
  { local vars; call A(Count(queryGS)); }
}.
```

Figure 16: Games required by `o2h` tactic. The local variable declaration `local vars` can be omitted (but then must be omitted in all games).

The tactic `o2h` can then be applied to proof goals of the exact form:

```
abs ( Pr[b=1 : left(rho)] - Pr[b=1 : right(rho)] )
        <= 2 * sqrt( (1+real q) * Pr[found : find(rho)] )
```

where `left` and `right` are the games from Figure 16 and `q` is an expression (of type `nat`). (This is exactly the shape of the conclusion of Theorem 3.)

When applying the tactic `o2h` (without any additional arguments), it checks whether all involved games have the right form and that none of the variables `count,found,G,H,S,in_S` are in the free variables of A (but A is allowed to access `IN,OUT,b,z`). If these checks succeeds, the tactic produces four subgoals:

```
Pr[count ≤ q : left(rho)] = 1
Pr[count ≤ q : right(rho)] = 1
Pr[count ≤ q : find(rho)] = 1
∀S G H z x. (S,G,H,z) ∈ supp distr → x ∉ S → G x = H x
```

The first three of them express the requirement that A makes at most q oracle queries (recall that `count` counts the oracle queries because of the wrapper oracle `Count`). And the fourth one

```
program queryG := {
  on IN, OUT apply (Uoracle G);
}.

program queryGS := {
  in_S <- measure IN with binary_measurement (proj_classical_set S);
  if (in_S=1) then found <- True; else skip;
  call queryG;
}.

program queryGM := {
  if (count=stop_at) then
    guess <- measure IN with computational_basis;
  else
    skip;

  call queryG;
}.

program Count(O) := {
  call O;
  count <- count + 1;
}.

program left := {
  count <- 0;
  (S,G,z) <$ distr;
  found <- False;
  { local vars; call A(Count(queryGS)); }
}.


program right := {
  count <- 0;
  stop_at <$ uniform {..<q};
  (S,G,z) <$ distr;
  { local vars; call A(Count(queryGM)); }
}.
```

Figure 17: Games required by `semiclassical` tactic. The local variable declaration `local vars` can be omitted (but then must be omitted in all games).

expresses the fact that $\forall x \notin S, G(x) = H(x)$ when S, G, H are chosen according to `distr`. (This is one of the premises of the O2H Theorem.)

Semiclassical-search tactic `semiclassical`. To invoke Theorem 4, we use the tactic `semiclassical`. This tactic requires that games of the exact form as in Figure 17 are defined. (Again, the names of the games, as well as the variables (IN, OUT, G, S, H, z, in_S, found, count, stop_at, guess) can be arbitrary, and the output type of G must be of type class `xor_group`. `distr` and `q` are arbitrary constant expressions.) See the description of the tactic `o2h` for programs `queryG`, `queryGS`, `Count`. The program queryGM is an oracle that first checks whether the number of the current oracle query is `stop_at` before querying G. If so, the input to G is measured in the computational basis and stored in `guess`. This corresponds to the query performed by the adversary B in Theorem 4. (Where `stop_at` is i in B.) And finally, the game `left` is like the `find` game in tactic `o2h` (i.e., lhs of the conclusion of Theorem 4) while the right game is rhs of the conclusion of Theorem 4.

Then the tactic `semiclassical`, invoked without any arguments, expects a subgoal of the

form:

```
Pr[found : left(rho)]
      <= 4 * real q * Pr[guess∈S : right(rho)]
```

It checks whether all games are as in Figure 17 and whether the free variables of A contain none of `G, S, H, in\_S, found, count, stop\_at, guess` (but A may access `IN, OUT, z, b`). If so, the tactic produces the following new subgoals:

```
Pr[count ≤ q : left(rho)] = 1
Pr[count ≤ q : right(rho)] = 1
```

Here `q` is the same expression as in the definition of program `queryGM` and `right`. These subgoals express the fact that the adversary makes at most q oracle queries.

References

[1] A. Ambainis, M. Hamburg, and D. Unruh. "Quantum Security Proofs Using Semi-classical Oracles". In: *Crypto 2019*. Springer, 2019, pp. 269–295.

[2] A. Ambainis, A. Rosmanis, and D. Unruh. "Quantum Attacks on Classical Proof Systems (The Hardness of Quantum Rewinding)". In: *FOCS 2014*. IEEE, 2014, pp. 474–483.

[3] F. Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019), pp. 505–510.

[4] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. "Computer-Aided Security Proofs for the Working Cryptographer". In: *Crypto 2011*. Vol. 6841. LNCS. Springer, 2011, pp. 71–90.

[5] G. Barthe, B. Grégoire, Y. Lakhnech, and S. Zanella Béguelin. "Beyond Provable Security Verifiable IND-CCA Security of OAEP". In: *CT-RSA 2011*. Vol. 6558. LNCS. Springer, 2011, pp. 180–196.

[6] G. Barthe, B. Grégoire, and S. Zanella Béguelin. "Formal Certification of Code-Based Cryptographic Proofs". In: *POPL 2009*. ACM, 2009, pp. 90–101.

[7] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. *CryptHOL: Game-based Proofs in Higher-order Logic*. IACR ePrint 2017/753. 2017.

[8] M. Bellare and P. Rogaway. "Optimal Asymmetric Encryption". In: *Eurocrypt '94*. Vol. 950. LNCS. Springer, 1994, pp. 92–111.

[9] M. Bellare and P. Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *CCS '93*. ACM, 1993, pp. 62–73.

[10] M. Bellare and P. Rogaway. "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs". In: *Eurocrypt 2006*. Vol. 4004. LNCS. Springer, 2006, pp. 409–426.

[11] M. Berg. "Formal verification of cryptographic security proofs". PhD thesis. Saarland University, 2013.

[12] B. Blanchet. "A Computationally Sound Mechanized Prover for Security Protocols". In: *Security and Privacy*. IEEE, 2006, pp. 140–154.

[13] D. Boneh, O. Dagdelen, M. Fischlin, A. Lehmann, C. Schaffner, and M. Zhandry. "Random oracles in a quantum world". In: *Asiacrypt 2011*. Springer, 2011, pp. 41–69.

[14] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé. *CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM*. IACR ePrint 2017/634. 2017.

[15] R. Canetti, O. Goldreich, and S. Halevi. “The Random Oracle Methodology, Revisited”. In: *STOC 1998*. ACM, 1998, pp. 209–218.

[16] J. F. Clauser, M. A. Horne, A. Shimony, and R. A. Holt. “Proposed Experiment to Test Local Hidden-Variable Theories”. In: *PRL* 23 (15 1969), pp. 880–884.

[17] A. W. Dent. “A Designer’s Guide to KEMs”. In: *Cryptography and Coding*. Berlin, Heidelberg: Springer, 2003, pp. 133–151.

[18] E. Fujisaki and T. Okamoto. “How to enhance the security of public-key encryption at minimum cost”. In: *IEICE Transaction of Fundamentals of Electronic Communications and Computer Science* E83-A.1 (Jan. 2000), pp. 24–32.

[19] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. “RSA-OAEP Is Secure under the RSA Assumption”. In: *Crypto ’01*. Vol. 2139. LNCS. Springer, 2001, pp. 260–274.

[20] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. “RSA-OAEP Is Secure under the RSA Assumption”. In: *J Cryptology* 17.2 (2004), pp. 81–104.

[21] D. Hofheinz, K. Hövelmanns, and E. Kiltz. “A Modular Analysis of the Fujisaki-Okamoto Transformation”. In: *TCC 2017*. Springer, 2017, pp. 341–371.

[22] K. Hövelmanns, E. Kiltz, S. Schäge, and D. Unruh. *Generic Authenticated Key Exchange in the Quantum Random Oracle Model*. IACR ePrint 2018/928, revision from February 14, 2019, `https://eprint.iacr.org/2018/928/20190214:124222`. Preliminary full version of [23]. 2019.

[23] K. Hövelmanns, E. Kiltz, S. Schäge, and D. Unruh. “Generic Authenticated Key Exchange in the Quantum Random Oracle Model”. In: *PKC 2020*. Vol. 12111. LNCS. 2020, pp. 389–422.

[24] A. Inoue, T. Iwata, K. Minematsu, and B. Poettering. “Cryptanalysis of OCB2: Attacks on Authenticity and Confidentiality”. In: *Crypto 2019*. Springer, 2019, pp. 3–31.

[25] ISO. *Information Technology – Security techniques – Authenticated encryption, ISO/IEC 19772:2009*. International Standard ISO/IEC 19772. 2009.

[26] M. Naehrig et al. *FrodoKEM*. Tech. rep. 2017.

[27] T. Nipkow. *Programming and Proving in Isabelle/HOL*. `https://isabelle.in.tum.de/website-Isabelle2019/dist/Isabelle2019/doc/prog-prove.pdf`. Version for Isabelle 2019. 2019.

[28] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.

[29] NIST. *Post-quantum crypto standardization – call for proposals*. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/call-for-proposals-2016.html`. 2016.

[30] A. Petcher and G. Morrisett. “The Foundational Cryptography Framework”. In: *POST 2015*. Springer, 2015, pp. 53–72.

[31] P. Rogaway. “Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC”. In: *Asiacrypt 2004*. Springer, 2004, pp. 16–31.

[32] T. Saito, K. Xagawa, and T. Yamakawa. “Tightly-Secure Key-Encapsulation Mechanism in the Quantum Random Oracle Model”. In: *Eurocrypt 2018*. Springer, 2018, pp. 520–551.

[33] P. W. Shor. “Algorithms for Quantum Computation: Discrete Logarithms and Factoring”. In: *FOCS 1994*. IEEE, 1994, pp. 124–134.

[34] V. Shoup. “OAEP Reconsidered”. In: *J Cryptology* 15.4 (2002), pp. 223–249.

[35] V. Shoup. *Sequences of games: a tool for taming complexity in security proofs*. Cryptology ePrint 2004/332. 2004.

[36] E. E. Targhi and D. Unruh. "Quantum Security of the Fujisaki-Okamoto and OAEP Transforms". In: *TCC 2016-B*. Vol. 9986. LNCS. Springer, 2016, pp. 192–216.

[37] D. Unruh. GitHub, `https://github.com/dominique-unruh/hksu-verification`. Source code of the proofs described here. 2020.

[38] D. Unruh. *dominique-unruh/qrhl-tool: Proof assistant for qRHL*. GitHub, `https://github.com/dominique-unruh/qrhl-tool`. Binaries of the correct version are at `https://github.com/dominique-unruh/qrhl-tool/releases/tag/v0.5`. 2017-2020.

[39] D. Unruh. *Local Variables and Quantum Relational Hoare Logic*. arXiv:2007.14155 [cs.LO]. 2020.

[40] D. Unruh. "Non-interactive zero-knowledge proofs in the quantum random oracle model". In: *Eurocrypt 2015*. Vol. 9057. Springer, 2015, pp. 755–784.

[41] D. Unruh. "Quantum position verification in the random oracle model". In: *Crypto 2014*. Vol. 8617. LNCS. Springer, 2014, pp. 1–18.

[42] D. Unruh. "Quantum Proofs of Knowledge". In: *Eurocrypt 2012*. Vol. 7237. LNCS. Springer, 2012, pp. 135–152.

[43] D. Unruh. "Quantum relational Hoare logic". In: *Proc. ACM Program. Lang.* 3 (Jan. 2019). Proceedings of POPL 2019, 33:1–33:31.

[44] D. Unruh. "Revocable quantum timed-release encryption". In: *J ACM* 62.6 (2015), 49:1–49:76.

[45] J. Watrous. "Zero-Knowledge against Quantum Attacks". In: *SIAM J Comput* 39.1 (2009), pp. 25–58.

[46] M. Zhandry. "How to Construct Quantum Random Functions". In: *FOCS 2013*. IEEE, 2012, pp. 679–687.

[47] M. Zhandry. "Secure Identity-Based Encryption in the Quantum Random Oracle Model". In: *Crypto 2012*. Vol. 7417. LNCS. Springer, 2012, pp. 758–775.