Developing a Quantum Crypto Theorem Prover from Scratch

Dominique Unruh
RWTH Aachen, Germany
University of Tartu, Estonia

Abstract

We describe our experience developing qrhl-tool, a theorem prover for verifying quantum cryptographic protocols, both in the post-quantum and the full quantum setting. The tool is built around quantum relational Hoare logic (qRHL), a relational program logic for reasoning about pairs of quantum programs in the style of game-based cryptographic proofs. We discuss the design choices underlying the tool: in particular, a hybrid architecture that delegates ambient-logic reasoning to Isabelle/HOL while qrhl-tool itself handles qRHL judgments and the program language; an advanced memoization mechanism (*hashed computations*) that enables efficient incremental proof checking; and a deliberate path towards a foundational implementation. We walk through a small worked example (the hardness of inverting $f \circ f$ given a one-way permutation f) to illustrate how these pieces fit together in practice. Along the way we highlight what we got right, what we got wrong, and which limitations (procedure parameters, local variables, runtime reasoning) we would approach differently if starting over today.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Cryptographic protocols; Theory of computation → Quantum complexity theory

Keywords and phrases Formalized mathematics, functional analysis, bounded operators

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.2

Category Invited Talk

Supplementary Material
Software (Source code): `https://github.com/dominique-unruh/qrhl-tool` [22]
archived at `swh:1:dir:e9a743ff12301274fa8ef17a2c36c6920f476bea`

Declaration about GenAI/LLM use Generative AI was used for proofreading the paper, for collecting bibliographic references (the references were all manually checked), and for drafting the abstract.

1 Introduction

Cryptographic systems are ubiquitous in today's society, and security flaws in cryptographic systems can have high costs, financially, in terms of privacy, or sometimes even human safety. Unfortunately, however, cryptographic systems are notoriously hard to analyze since we need to prove not only that they withstand normal operations, but that they are secure under any adversarial behavior. Especially if we are in the *computational model*, the proofs involve complexity-theoretic arguments, involving nontrivial arguments about the runtime of adversaries. (The computational model of cryptography considers the actual computational problems involved in an attack, such as needing to factor large integers; in contrast, the symbolic model is an idealized model that abstracts from these.) This creates the situation where, even if we have a mathematical proof of security, we may have limited trust in the security of the system. After all, human-generated proofs may have undetected accidental mistakes, and the more complex the proof, the more likely this is.

As a consequence, mechanized verification of cryptographic protocols was introduced. Most famous, probably, is the EasyCrypt tool [5, 3]; however, a number of other tools have also approached this task, such as CryptoVerif [11], CertiCrypt [4], FCF [20], CryptHOL [6], and SSProve [17]. The typical approach is to treat the cryptographic scheme and the security definition together as a small program, called a "game" [21, 8], and use techniques from program analysis to show the desired security properties.

The situation becomes even more dire when considering quantum cryptography. Quantum cryptography comes in two flavors: Post-quantum cryptography considers the security of classical protocols against adversaries that have a quantum computer (quantum adversaries for short). Quantum protocols, in contrast, actually involve quantum-mechanical processes in the construction of the protocol. (That is, even honest protocol participants have to use quantum communication channels or quantum computers.) Quantum cryptographic security proofs, especially in the computational model, are even more involved, and, especially due to the non-intuitive nature of quantum mechanics, more likely to contain incorrect assumptions and mistakes.

In this light, it is only natural to want a way to do mechanized verification of quantum cryptography. (Both post-quantum cryptography and quantum protocols.)

In this paper, we describe our solution to this problem, the qrhl-tool quantum cryptography theorem prover [30, 24]. We describe the overall structure, some of the challenges encountered, and discuss some of our experiences with the choices we made.

Related work. There is a second theorem prover (developed subsequently) for verification of post-quantum crypto, namely EasyPQC [2]. EasyPQC takes a simpler approach to the underlying logic; it can only explicitly express programs that compute classical functions in superposition, and pre- and postconditions are classical (and hold in superposition). This is probably enough for many post-quantum crypto proofs, but not all. However, we should stress that the implementation of EasyPQC does not match the theory in the paper, and is not sound.[1]

2 How to do (quantum) crypto proofs

Before we start discussing our theorem prover itself, let us take a step back and understand how proofs are usually done in cryptography. For this, we consider a simple example, the one-way permutation. A one-way permutation is a bijective function f that is easy to compute, but for which it is hard to compute inverses. The cryptographic security property here is the "hard to compute inverses" part; it talks about something an adversary should not be able to do. Of course, a sufficiently powerful adversary could just enumerate all potential preimages of y to find x with $f(x) = y$, so a one-way permutation can only be secure against computationally limited adversaries. This is where the complexity-theoretic aspects of cryptography come from. We can define this security property as follows:

▶ **Definition 1.** *A bijective function $f : X \to X$ is (τ, ϵ)*-one-way *iff for any τ-time adversary A, we have* $\Pr[x = x' : x \xleftarrow{\$} X, y \leftarrow f(x), x' \leftarrow A(y)] \leq \epsilon$.

Here $\Pr[x = x' : x \xleftarrow{\$} X, y \leftarrow f(x), x' \leftarrow A(y)]$ denotes the probability that $x = x'$ holds after executing the following steps: Pick x uniformly at random from X, set $y := f(x)$, and then run adversary A with input y. So altogether, it means: what's the probability

[1] Based on our own experiments and discussions with the authors.

that the adversary finds the preimage x, given $y = f(x)$. And we want it to be $\leq \epsilon$, at least if the adversary doesn't run more than τ time steps. The values τ (runtime) and ϵ (attack probability) are parameters of the definition, they tell us how good of a one-way permutation f is.

How can we prove anything about this? A simple example:

▶ **Theorem 2.** *If f is (τ, ϵ)-one-way, then $g := f \circ f$ is $(\tau - \tau_f, \epsilon)$-one-way where τ_f is the time needed for one evaluation of f.*

Proof. Let A be a $(\tau - \tau_f)$-time adversary. Let B be the adversary defined by $B(y) := f(A(y))$. Notice that B is a $((\tau - \tau_f) + \tau_f)$-time adversary, i.e., τ-time. We have

$$\begin{aligned}
&\Pr[x = x' : x \xleftarrow{\$} X, y \leftarrow g(x), x' \leftarrow A(y)] \\
&= \Pr[x = x' : x \xleftarrow{\$} X, z \leftarrow f(x), y \leftarrow f(z), x' \leftarrow A(y)] \\
&= \Pr[z = f(x') : x \xleftarrow{\$} X, z \leftarrow f(x), y \leftarrow f(z), x' \leftarrow A(y)] \\
&= \Pr[z = z' : x \xleftarrow{\$} X, z \leftarrow f(x), y \leftarrow f(z), z' \leftarrow B(y)] \\
&= \Pr[z = z' : z \xleftarrow{\$} X, y \leftarrow f(z), z' \leftarrow B(y)] \\
&\leq \epsilon
\end{aligned}$$

Here all equalities are obvious since we only replaced the programs and conditions in the probabilities by programs and conditions that do the same. And $\leq$ holds by assumption because f is one-way and B is τ-time. ◀

This very simple example illustrates a number of crucial aspects of cryptographic proofs:

- *Computational assumptions:* We don't show unconditionally that our cryptosystem (here g) is secure, but instead show it under the assumption that something else is computationally difficult (here breaking f). This is state of the art; the kinds of assumptions made vary drastically.
- *Proof by reduction:* The proof involves a reduction: we define a new adversary B in terms of the adversary A we are actually interested in. This adversary (after a more or less complex sequence of transformations) is then considered as an adversary against our computational assumption.
- *Reduction losses:* In our example, the security of g is worse than that of f. Namely, g is only secure against $(\tau - \tau_f)$-adversaries. Similarly, in most proofs the ϵ will also change.
- *Runtime calculations:* We need to know the runtime of B given the runtime of A. In general, these calculations can be more complex.
- *Sequences of games / program transformations.* A large part of the proof consists of starting with one small program (the operations $x \xleftarrow{\$} X, y \leftarrow g(x), x' \leftarrow A(y)$ from the proof of Theorem 2), and transforming it step by step until we reach something easier to analyze (last probability in the proof). These small programs are traditionally called "games", hence the name "game-based proofs" for this style of proofs [21, 8].

We cheekily called the transformation steps "obvious" in our proof. In reality, they may only seem obvious because we have an intuition about how to read algorithms. Yet a cryptographer will often be hard pressed to nail down the exact mathematical laws involved here. This is aggravated by the fact that in pen-and-paper cryptography, there is rarely a precise definition of the underlying "programming language" and its semantics. This is one of the avenues where a mechanized proof will differ most from the pen-and-paper one.

A second issue is the runtime calculations: We claimed that B has an additive overhead of τ_f steps compared to A, where τ_f is the time for computing f. This may seem obvious since B only has one additional computation of f. But without a precise machine model we lack a rigorous argument for that. In fact, cryptographers will usually not specify the machine model, or claim that the machine model is that of a Turing machine without the details.[2]

Quantum cryptography. The above discussion was a bit vague as to whether we were speaking about classical or quantum cryptography. In fact, the above example can be applied the same way to a classical and to a quantum crypto setting. Namely, it just boils down to whether we interpret a "τ-time adversary" as meaning a "τ-time probabilistic algorithm" or a "τ-time quantum algorithm". In the first case, we will have a classical crypto definition and proof, in the second case a quantum crypto definition and proof. (More specifically, we are then doing *post-quantum* crypto as the function f that would be used by honest participants is still a classical function.)

But wait, how can this be? How can the same proof show both classical and quantum security? Isn't quantum security a stronger requirement (since adversaries with a quantum computer are more powerful than those without)? The answer lies in the assumptions we make: If we interpret Theorem 2 in the quantum setting, what it says is "if f is secure against quantum adversaries, then g is secure against quantum adversaries". So this is an incomparable statement as we have strengthened both the premise and the conclusion. This is often the case: when going to the post-quantum setting, all we need to do is to reinterpret the assumption to hold against quantum adversaries.

This example may lead to the conclusion that when doing quantum cryptography, proofs don't change. The adversary may be quantum, but that is hidden away from our sight since we only use it as a black box (e.g., when wrapping A inside B). This is not (always) true, though:

- If we are considering quantum protocols (where also the honest protocol participants use quantum mechanics), there is no way around understanding quantum mechanics.
- Rewinding is a crypto proof technique where we use programs such as "run adversary A, then reset A to an earlier state, then run A again". While this sounds innocuous (all we need is a memory snapshot at the right time), it fails in the quantum setting because it is not possible to clone quantum states. Very different (and much more complex) proof techniques are needed [37, 29].
- Random oracles [7] are an idealized modeling of hash functions. We omit the details here, but it suffices to know that in the quantum setting, random oracles inherently talk about quantum-mechanical effects [13].
- Last but not least – even in the simple case above, if we go for full rigor (or mechanized proofs), the definition of the semantics of programs, and the justification under the hood for the different reasoning rules, will be inherently quantum. Our simple proof just doesn't show this.

[2] Using Turing machines is probably worse than not specifying a machine model at all: Turing machines do not match our intuition for how computers work very well. For example, we are not aware of crypto papers that would include the time for moving the Turing machine heads into the right position, or clearing a tape for invoking A after computing f. We make the daring claim that cryptographers never really mean Turing machines when they talk about Turing machines, but rather some variant of RAM machines.

3 The underlying logic

As the aforementioned section has illustrated, at the core of classical or quantum crypto proofs is the concept of a sequence of games where we need to relate consecutive programs. We therefore need some logical foundations that help us to talk about the relationship of two programs. In the simplest case, we might want to say something like "both programs compute the same function", but in many cases (especially inside a proof when relating individual program fragments), we may need to be more flexible and express arbitrary relationships between the inputs and outputs. This can be done with so-called relational Hoare logic (RHL) [9]. In RHL, we can compare two deterministic imperative programs p and q, and make statements such as "if the variables of p and q stand in relation A before running the programs, then after running the programs, they stand in relation B." Written, e.g., $\{A\}\, p \sim q\, \{B\}_{\mathsf{RHL}}$. An example would be:

$$\{x_1 = x_2\}\, x \leftarrow x + 1 \sim x \leftarrow x - 1\, \{x_1 = x_2 + 2\}_{\mathsf{RHL}}$$

The formal definition is fairly straightforward:

▶ **Definition 3** (RHL). *$\{A\}\, p_1 \sim p_2\, \{B\}_{\mathsf{RHL}}$ holds iff for any two initial states m_1, m_2 that satisfy A, we have that m_1', m_2' satisfy B where $m_i' := [\![p_i]\!](m_i)$.*

To be fully precise, we of course additionally need to define the denotational semantics $[\![p]\!]$ of a deterministic program mapping the initial state m to the final state m'.

As an important special case, we can express that two programs are denotationally equal as $\{x_1 = x_2 \wedge y_1 = y_2 \wedge \dots\}\, p_1 \sim p_2\, \{x_1 = x_2 \wedge y_1 = y_2 \wedge \dots\}_{\mathsf{RHL}}$ where $x, y, \dots$ are all variables of the programs p_1, p_2.

This concept was generalized by [4] to the case of probabilistic programs. In probabilistic relational Hoare logic (pRHL) we compare two probabilistic programs, and we still wish to state that some relationship between their variables holds after execution. One might be tempted to say:

▶ **Definition 4** (pRHL – bad variant). *$\{A\}\, p_1 \sim p_2\, \{B\}_{\mathsf{pRHL\text{-}bad}}$ holds iff for any two initial states m_1, m_2 that satisfy A, we have that m_1', m_2' satisfy B with probability 1, where the m_i' are sampled (independently) according to the distributions $[\![p_i]\!](m_i)$. ($[\![p]\!](m)$ is the denotational semantics that returns the distribution of final states.)*

This definition is not satisfactory. For example, $\{x_1 = x_2\}\, x \xleftarrow{\$} \{0,1\} \sim x \xleftarrow{\$} \{0,1\}\, \{x_1 = x_2\}_{\mathsf{pRHL\text{-}bad}}$ does not hold. Namely, if we sample the final states, we get a random bit x_1 and a random bit x_2, and these will be equal only with probability $\frac{1}{2}$. Yet, we would want this to hold since the two programs are equal. (Otherwise it will become very hard to reason about the fact that two programs do the same).

The solution from [4] is:

▶ **Definition 5** (pRHL). *$\{A\}\, p_1 \sim p_2\, \{B\}_{\mathsf{pRHL}}$ holds iff for any two initial states m_1, m_2 that satisfy A, we have that: There exists a joint distribution μ_{12}' on program states such that $[\![p_1]\!](m_1), [\![p_2]\!](m_2)$ are the left/right marginals of μ_{12}', and such that m_1', m_2' jointly sampled from μ_{12}' satisfy B.*

In other words, if we sample $(m_1', m_2') \leftarrow \mu_{12}'$, then m_i' is distributed like the final state of $[\![p_i]\!](m_i)$, but they are not necessarily independent. This logic behaves a lot better than pRHL-bad; this stems from the fact that sampling m_1, m_2 independently in Definition 4 led to a much too strict interpretation of satisfying the postcondition. In particular, we now have $\{x_1 = x_2 \wedge y_1 = y_2 \wedge \dots\}\, p_1 \sim p_2\, \{x_1 = x_2 \wedge y_1 = y_2 \wedge \dots\}_{\mathsf{pRHL}}$ iff $[\![p_1]\!] = [\![p_2]\!]$.

This definition turns out to give rise to a logic that is very useful for reasoning about classical cryptographic sequences of games, as shown in particular by the success of the EasyCrypt tool [5] which uses it as its main logic.

Quantum relational Hoare logic. Inspired by pRHL, we developed an analogous logic for the quantum setting, quantum relational Hoare logic (qRHL). This faced additional challenges beyond those faced in the probabilistic case. Depending on small changes in the design decisions (that made no difference for the probabilistic case), different reasoning rules failed and led to an unusable logic. A thorough discussion is given in [30]; we only present the final result here. We will try to describe the definitions in a way that does not require expertise in quantum computing, at the expense of being informal.

The first question is how predicates on initial/final states can be expressed. In the deterministic and probabilistic case, the predicates A, B were simply logical expressions[3] involving the variables $x_1, y_1, \dots$ and $x_2, y_2, \dots$ of the two states m_1, m_2. Or, stated differently, a predicate is represented as a set of pairs of memories. By "memory" we mean an assignment of values to all program variables. Let $\mathcal{M}$ denote the set (or type) of all memories, formally something like $\mathcal{M} := \prod_{x \in Vars} \mathsf{Type}(x)$.

In the quantum case, this is not as simple, since the state of a program (before or after running it) can be a superposition of many different variable assignments, e.g., $\alpha_1|m^{(1)}\rangle + \alpha_2|m^{(2)}\rangle + \dots$ where $m^{(i)} \in \mathcal{M}$ are different classical states, and the α_i tell us how much each classical state is present in the superposition (and the $|\dots\rangle$ is some customary notation). Intuitively, one can think of such a state as a vector with coefficients α_i. All possible quantum states over $\mathcal{M}$ of a program form a Hilbert space that we call $\ell_2(\mathcal{M})$. (A Hilbert space is a vector space with some additional properties.)

A natural way of representing predicates over quantum states is then not to specify an arbitrary set of quantum states (as in the classical case), but specifically a *subspace* of the Hilbert space of all states. (This is also known as Birkhoff–von Neumann quantum logic [10].) Specifically, when talking about the relationship of states of two programs, we use subspaces of $\ell_2(\mathcal{M} \times \mathcal{M})$.

We then say two quantum states ψ_1, ψ_2 satisfy a predicate A if their tensor product $\psi_1 \otimes \psi_2 \in A$. (The tensor product $\otimes$ puts two quantum states on smaller quantum systems together into a bigger system.) We will also need to consider so called mixed states (density operators) ρ, for the sake of this exposition it suffices to know that these essentially represent probability distributions over quantum states, needed for representing the output of programs that involve measurements and classical probabilistic operations.

With this notion, we are ready to define qRHL, quite analogous to pRHL:

▶ **Definition 6.** *For quantum programs p_1, p_2,[4] $\{A\}\, p_1 \sim p_2\, \{B\}_{\mathsf{qRHL}}$ holds iff for any two initial states ψ_1, ψ_2 with $\psi_1 \otimes \psi_2 \in A$, we have that: There exists a joint separable mixed state distribution ρ'_{12} over $\mathcal{M} \times \mathcal{M}$ such that the mixed states $[\![p_1]\!](\psi_1), [\![p_2]\!](\psi_2)$ are the left/right marginals of ρ'_{12}, and such that ψ'_1, ψ'_2 jointly sampled from ρ'_{12} satisfy $\psi'_1 \otimes \psi'_2 \in B$.*

[3] Which logic, like first-order logic, higher-order logic, etc., depends on the specific implementation.

[4] Technically, we would need to define exactly what programs we can represent, and their semantics. We omit this in this exposition. Suffice it to say that it's a programming language with classical and quantum variables, unitary operations, if- and while-statements, and probabilistic sampling. Semantically, $[\![p]\!]$ is a function mapping mixed states to mixed states.

Here the "left/right marginals" of ρ'_{12} are the states one gets when throwing away the right/left half of the state ρ'_{12} (partial trace). And "separable" means that there is no quantum entanglement between the left and right part of the state. (The requirement of separability has no intuitive justification. This is simply one of the variations of the qRHL definitions we needed to fine-tune to make reasoning work in the end.)

Syntax for predicates. Designing the basic meaning of qRHL judgments (Definition 6) is only part of what we need to do for the logical foundations. For example, we need to decide how to actually write predicates. In the classical case, even though predicates are logically sets, we prefer to write them as a first- or higher-order formula over variable names. Similarly, in the quantum case, we don't want to specify a subspace by explicitly specifying a basis to the subspace, we want to be able to describe it via formulas. Fortunately, this turns out to be reasonably easy: Operations such as sum and intersection of subspaces take (vaguely) the role of disjunction and conjunction in predicate logic, and even quantifiers have their analogues in suprema and infima of families of subspaces. Adding some basic predicates such as $x =_{\mathsf{q}} \psi$, denoting the subspace of all states where the variable x has state ψ, we can write quite powerful predicates. For example, $\bigvee_{a\in\mathbb{N}} x =_{\mathsf{q}} |a^2\rangle$ would mean "there exists an a such that x is in state $|a^2\rangle$." Particularly interesting is the question how to express "x_1 and x_2 have the same quantum state" (something that in the classical setting is simply $x_1 = x_2$). We will not go into details; it suffices to say that there is a meaningful way to express this as a subspace, which we write $x_1 \equiv_{\mathsf{q}} x_2$. With this definition, we can recover important reasoning rules from the classical case. For example, $\{(x_1, y_1, \dots) \equiv_{\mathsf{q}} (x_2, y_2, \dots)\}\, p_1 \sim p_2\, \{(x_1, y_1, \dots) \equiv_{\mathsf{q}} (x_2, y_2, \dots)\}_{\mathsf{qRHL}}$ iff $[\![p_1]\!] = [\![p_2]\!]$. It should be noted, however, that this quantum equality does not always behave exactly like classical equality does; it does have its occasional quirks. For details, see [30].

Local variables. An additional question to cover is that of procedures, parameters, return values, and local variables. After all, we need to be able to declare an adversary, and call the adversary from another program, and define reduction adversaries, and call encryption schemes, and much more. Unfortunately, in the quantum setting, these things are more complex. We cannot simply pass the values of quantum variables as this would mean copying quantum data which is not meaningful. Instead, we need to do some kind of call-by-name, then worry about aliasing etc. We decided to try and avoid these complications in our logical foundations, and have opted for a simplistic solution to this:

There are no procedures, only imperative programs without parameters or return statements acting on explicitly declared global variables. The only language feature we add is the possibility that one program calls another (without passing inputs or getting outputs back). How can we use that to encode statements like $x' \leftarrow A(y)$? We simply declare global variables `adv_in` and `adv_out`, and we write $\texttt{adv\_in} \leftarrow y, \texttt{call}\ A, x' \leftarrow \texttt{adv\_out}$. And the adversary accesses `adv_in` or `adv_out` directly.

We then claimed: besides somewhat annoying manual bookkeeping about which variable is allowed to be accessed by which program (which we have to support anyway to allow global variables with secrets), we can then emulate anything that can be done with proper procedure calls and local variables. (Recursive calls are something you cannot emulate this way, but those are typically not needed in the programs we encounter in game-based proofs.) It is admittedly somewhat annoying for the user, but no real limitation considering that the programs we consider are relatively small.

Unfortunately, we were wrong. While this argument is sound in the classical case, it does not work in the quantum case. The difference between our "pseudo-local" variables (i.e., global variables that we mentally declare as local) and true local variables is that for true local variables, it is guaranteed that their content is destroyed after a procedure exits. Keeping the value of a variable around seems irrelevant as long as we don't refer to it anymore (in programs or post-conditions). Yet, quantum equality has an idiosyncrasy: $x_1 \equiv_{\mathsf{q}} x_2$ can only hold if x_1, x_2 are not entangled with any other variables. So if there is a "pseudo-local" variable y still lying around that got entangled with x_1 at some point, $x_1 \equiv_{\mathsf{q}} x_2$ becomes untrue due to the mere existence of y.

So we ended up extending our logical foundations with local variables in a somewhat ad-hoc manner. We introduce a local block $\{\texttt{local}\ x; P\}$. This means (informally speaking), that the value of the *global* variable x is pushed onto a stack, and then initialized within the block with a fresh value. And at the end of the block it is recovered from the stack. See [31] for details. This works ok, but leads to annoying corner-cases when not used carefully. In particular, if P calls other programs, the local-block around P may actually affect their access to x, too. In a nutshell, we are not very happy with this solution; in retrospect we would have done it differently. But we stuck to this solution because a different approach would have required a major rewrite of the theorem prover at that point. (But see Section 6.)

4 Architecture

We now discuss the overall architecture of our theorem prover. That is, design choices that are not very specific to qRHL and quantum crypto, but may inform the design of other special purpose theorem provers.

4.1 Hybrid architecture

One probably uncommon design choice we made is a hybrid architecture involving our self-written tool (in Scala), and the existing Isabelle/HOL theorem prover. The motivation came from our experiences with EasyCrypt: After a first attempt at embedding pRHL directly in the Rocq theorem prover [4], they decided that it would be less cumbersome and more user-friendly to implement a tool for reasoning about pRHL from scratch, called EasyCrypt. EasyCrypt has builtin support for probabilistic while programs and pRHL judgments and reasoning. This avoids the necessity of modeling all of this in Rocq (making life easier for the developers), and avoids possible idiosyncrasies of the embedding of pRHL into Rocq (making life easier for the user). There is, however, one drawback: In cryptographic proofs, we often do not need to purely apply pRHL reasoning rules, we also need to perform a fair degree of reasoning outside of pRHL. (To give a simple example: in an analysis of the RSA cryptosystem, we will need the various facts about modular arithmetic.) So all the necessary mathematics has to be formalized from scratch in EasyCrypt, even though it may already exist in other theorem provers that are better suited for general purpose math reasoning. This meant that EasyCrypt had to develop the necessary capabilities to support these proofs, too. This seemed like an unnecessarily large burden on the developers to us.

In the quantum setting, this is even more of a hindrance: Now we need to reason about Hilbert spaces, operators, subspaces, etc. So to avoid this, we designed our prover `qrhl-tool` with a hybrid architecture:

On the surface, qrhl-tool has its own input syntax and reasoning capability (tactic-based backward reasoning). However, we developed a Scala library `scala-isabelle` [28] that allows to start an instance of the Isabelle theorem prover in the background. Qrhl-tool does

this, and delegates all reasoning in the ambient logic to Isabelle/HOL. (By ambient logic we mean anything that does not involve qRHL judgments.) Specifically, we got the following advantages:

- We did not need to invent any syntax for formulas that are not specific to qRHL or the program language. Whenever we encounter some expression in ambient logic, we send it to the Isabelle process for parsing. For example, when we write `lemma name: 1+1=2.`, the proposition `1+1=2` is parsed by Isabelle, and its meaning is opaque[5] to our tool.
- We get large parts of the syntax for programs for free. For example, assignments have the form `x <- expr;` where `expr` is sent to Isabelle for parsing and can be an arbitrary well-typed Isabelle term. Same for more complex objects, such as `on x apply operator;` where `operator` can be an arbitrary mathematical description of an operator to apply to the quantum variable `x`.
- We get a syntax for predicates. Remember from Section 3 that predicates are subspaces, but that we want to specify them using various connectives (like supremum) to get a usable way to work with predicates. This is made easy:[6] We just define the various connectives as constants in Isabelle (or use existing constants), and export the parsing to Isabelle. When doing reasoning in qRHL, tactics will often copy the predicates more or less unchanged into the resulting verification conditions (e.g., `some_subspace` $\leq$ `some_other`). This works very smoothly with our approach, since both the predicates and the verification condition are internally already stored as Isabelle terms.
- We can reuse existing tactics from Isabelle and adapt them easily in our tool. For example, our `simp` tactic, when applied to a qRHL judgment, will run Isabelle's simplifier on the pre-/postcondition.
- We can import Isabelle theories. So if there is some complex mathematical lemma we need in our proof, and that lemma is not specific to qRHL or our programs, we write and prove that lemma completely inside Isabelle, and use it in the qrhl-tool proof.

However, we should not hide the disadvantages of this approach: The learning curve for the tool may increase (unless the user is already familiar with Isabelle/HOL) because now one needs to work in Isabelle/HOL and qrhl-tool, depending on the specific subtask. It can be annoying to have to switch between two tools repeatedly. And finally, setup becomes somewhat more complex because one needs to install both qrhl-tool and a matching Isabelle implementation (and configure the paths in qrhl-tool).

What alternatives are there to this approach?

- Write a theorem prover from scratch without using any preexisting prover. We discussed the disadvantages above.
- Implement the new theorem prover as a library of theorems and special purpose tactics. This approach was taken, e.g., by [6, 20, 17] for classical crypto. Whether this approach leads to more or less usability depends a lot on the underlying theorem prover and how customizable it is. E.g., can we introduce custom syntax for programs, etc.? Are there any logical constructs in program declaration that cannot easily be modeled?[7]

[5] Actually, in our tool we *can* inspect the internals of this term. But we don't have to when it does not involve qRHL specific reasoning.

[6] Easy from the programming point of view. There is still a fair amount of mathematics formalization involved, see [14, 36, 35].

[7] For example, if in qrhl-tool, we declare two variables via `quantum var x : bit. quantum var y : bit.`, they will automatically be different variables (`x` $\neq$ `y`). This is important, e.g., when trying to show that one subprogram (using `x`) does not interfere with another (using `y`). Similarly in EasyCrypt. In a typical math-oriented theorem prover, there is no immediate analogue to this. Two constants that are defined the same way (e.g., `consts x : bit qvariable` or similar) can never be shown to be distinct. So when embedding into a general purpose theorem prover, one will have to deal with questions like "how to show that two variables are not the same one" that follow by definition in a special-purpose theorem prover written from scratch.

We do not have a final answer as to whether the approach we took was the best for this situation, but it is certainly one to consider when writing another domain-specific theorem prover from scratch.

To conclude, we should also discuss why we specifically used Isabelle/HOL as the underlying theorem prover. At the time of starting the project (mid 2017), according to our best knowledge at the time, Isabelle/HOL had the best existing library for real and complex analysis, on which we based our mathematical foundations for quantum mechanics. We feel that this situation has since changed, with Lean's mathlib [18] providing a large amount of foundations. So if we were to start the project again, we might choose Lean [15] instead.

Another reason for Isabelle was the availability of the sledgehammer tool [12] which is very good at proving simple subgoals automatically. This advantage needs to be revisited; we are not sure about the state-of-the-art in Lean in that respect.

We might also choose to avoid a hybrid architecture, and instead try to stay completely within Lean, trying to customize it as far as possible to make the tool easy to use.

4.2 Hashed computations

In a nutshell, the main-loop of the theorem prover reads commands one at a time (coming from the user, e.g., via the ProofGeneral [1] UI), keeps a list of all commands, and keeps a tree of all files in the current directory. Each time a command is added (or a command is removed via `undo`), a stateless function is called and passed the file tree and the command list, and computes the result of executing all commands in all dependencies and the command list.

Now this has the advantage of being quite simple conceptually: We do not need to track where we need to restart computation, when some dependency has changed. On the other hand, if done as described here, the prover would be horrendously inefficient; each time a command is entered, all proofs would be rechecked.

This is why we added another layer to this approach, a form of advanced memoization. When a function is called, a hash of all its inputs is computed, and the function is rerun only if the inputs changed. That, on its own, would already avoid rechecking all proofs. The computations that only depend on inputs that have not changed would not need to be recomputed. However, we can go further: we implemented a datastructure that keeps track of which parts of its inputs a memoized function accesses. If these parts have not changed, the function does not need to run again. For example, if a function takes a file system tree as input, but only accesses one file in it, it would only need to be recomputed if that file changes.

This advanced memoization leads to several advantages (without additional explicit tracking of dependencies):

- When a dependency changes, the changes will automatically be taken into account without needing to rerun anything explicitly.
- If the user edits a dependency, or the proof further up, but that change is purely presentational (whitespaces, comments, ...), nothing will be recomputed. (The parsing will happen again, since the memoization of the parse function will notice changed inputs. But the output of the parsing will not change, and therefore the proof checking, which is the computationally heavy stuff, will not happen again.)
- If the user edits a proof higher up or in a dependency, this will not require recomputation of everything between that proof and the current point, since nothing depends on the proof content, only on the proven theorem itself.

- When the user edits something (e.g., accidentally inserts a character near the beginning of the file), leading to recomputation, and then undoes that change, then we recover the previous state instantly (no recomputation needed). Without memoization, the user would in such a situation have to wait for the whole file to rerun.

We believe this approach might be useful in similar tools, too. (Theorem provers; compilers with incremental compilation.)

We have implemented this mechanism in an independent library `hashedcomputation` [22]. (However, we should stress that this library is not very polished; it is used only inside the theorem prover and would need additional work to be a production ready library.)

4.3 ProofGeneral

We used ProofGeneral [1] to provide the UI. In a nutshell, ProofGeneral allows us to use Emacs to edit a proof file, and to send it line by line to the prover (and to go back to earlier positions, creating synthetic *undo* commands). It also parses the output of the tool to present a message and a goal window.

With ProofGeneral, it is very easy to develop the UI of an interactive theorem prover (essentially, the theorem prover just needs to read commands one by one). To get things like semantic syntax highlighting, information when hovering over symbols, etc., as customary in modern IDEs, this simplistic approach is not suitable. A more structured protocol (such as language server protocol [19]) with a compatible IDE would probably have been a better starting point.

4.4 Path towards a foundational prover

A theorem prover is called foundational if all reasoning is broken down to elementary mathematical reasoning steps. A non-foundational one can have non-trivial tactics that are proven sound by hand, but require us to trust their implementation. Note that a foundational prover also can have powerful tactics on the user level; that tactic just needs to internally break down its operation to a sequence of elementary steps.

One additional reason for our hybrid architecture (not mentioned in Subsection 4.1) was to have a migration path towards a foundational theorem prover. Our theorem prover qrhl-tool currently has many tactics that correspond to rules proven secure, e.g., in [30], but not implemented foundationally. This was important to be able to get qrhl-tool up and running, but we would like to have the possibility to transition to a foundational approach in the long run. The hybrid architecture allows this: we can fill in more and more of the underlying math in Isabelle/HOL, and transfer one tactic at a time into Isabelle/HOL (while keeping the frontend the same). This makes the transition easier because one does not need to make a hard cut and restart from scratch.

This is still work in progress (with a lot of work ahead).

5 QRHL in qrhl-tool

We now explain – on a high level – how qRHL is supported in qrhl-tool. For this, we go through a simple example, and discuss on each step how the architecture supports this. (And what limitations there are.) The example is the example proof from Theorem 2. Admittedly, this is a very simple example that seems to hardly touch quantum-specific challenges. But since under the hood, the same machinery needs to be used even for these simple cases, it is still a convenient example. Details of the commands involved can be found in the qrhl-tool manual [26]. The source of the example can be found in [23].

As a first step, we create an Isabelle file `OWP.thy` that contains all declarations (and possibly lemmas) that we wish to do in Isabelle/HOL:

```
theory OWP imports QRHL.QRHL begin
declare_variable_type X
axiomatization f :: ‹X ⇒ X› where bij_f: ‹bij f›
definition g where ‹g = f ∘ f›
lemma [iff]: ‹f x = f y ↔ x = y› by [...]
end
```

In this file, we define a not-further specified type `X`; declare a function `f` and axiomatize that it's a bijection, and let `g` be `f` ∘ `f`. Mostly, this is standard Isabelle. Yet two things are specific to qrhl-tool:

- First, we load a background theory `QRHL.QRHL`; this theory loads all Isabelle formalizations we made to support qrhl-tool: both mathematical theories and definitions (more on that later), and project-specific Isabelle configuration.
- We declare the type `X` with a custom command `declare_variable_type`. This declares a new type, but adds qrhl-tool-specific customization: specifically, it declares it to be of typeclass `universe` which says that the type is small enough to be used as type of a program variable.[8]

Now the main qrhl-tool file `owp.qrhl`:

```
isabelle OWP.
```

This command loads the internal Isabelle process, with the theory `OWP` we just defined.

```
quantum var qglobA : infinite.
classical var cglobA : infinite.
classical var adv_in : X.
classical var adv_out : X.
classical var x : X.
classical var x' : X.
```

Here we declare all the program variables needed throughout the proof. (I.e., global variables accessible by all programs.) We can declare classical and quantum variables. Internally, not much is happening, they are just registered in an environment. The tool does, however, check that they are all of typeclass `universe`.

Let us understand what variables we need to declare, and how they relate to some of the limitations of qrhl-tool:

- `cglobA`, `qglobA`: We'll define an adversary later; this adversary is supposed to be a quantum adversary. Since the adversary is assumed to have a persistent global state, we create two program variables that "belong" to the adversary, one for classical and one for quantum state. They are of type `infinite` which is a predeclared infinite type; to model that the adversary has arbitrarily much memory. (We could as well have used `bit list` or similar; using `infinite` is just stylistic.)
- `adv_in`, `adv_out`: These are needed for inputs/outputs in adversary calls. This is necessary because of our lack of support for function parameters as discussed in Section 3, page 7.
- `x`, `x'`: These are just variables we will use in our programs below. If we had true local variables, we would not need to declare them as global variables here.

[8] Basically, any type that is constructed from countable types, powersets, and function types is small enough. But if we would allow arbitrarily large types, we would get into logical contradictions since the type of all program states would then be a valid variable type.

```
adversary A free qglobA, cglobA, adv_in, adv_out.
```

Our result needs to hold for any adversary `A`. So we declare one. Implementation-wise, this just means registering this adversary in the environment. Logically, it means `A` is an arbitrary fragment of code. The only restriction put on it is that it accesses only the variables `qglobA`, `cglobA`, `adv_in`, `adv_out`. This is important: if it could access `x`, it could simply read the secret value it needs to guess.

```
program g_owp := {
  x <$ uniform UNIV;
  adv_in <- g x;
  call A;
  x' <- adv_out;
}.
```

This is used to define what it means for `g` to be a one-way permutation. Notice how this is not a full definition. We simply define a program that corresponds to the fragment $x \xleftarrow{\$} X, y \leftarrow f(x), x' \leftarrow A(y)$ inside Definition 1. The full security definition would be something like "for any τ-time A, $\Pr[x = x' : \text{g\_owp}] \leq \epsilon$". This overall security definition is left implicit; we will talk about this limitation later. Internally, this just registers the program code in an environment. It also defines `g_owp` in the Isabelle process; this currently uses an embedding where the different program constructors (such as `assign`, `sample`, `while`, ...) are simply axiomatized constants. This means, Isabelle cannot prove anything about these programs, but it provides a path towards actually implementing those definitions later and migrating towards a foundational tool (see Subsection 4.4).

```
program B := {
  call A;
  adv_out <- f adv_out;
}.
```

This defines the adversary `B` in terms of `A` (see the proof of Theorem 2). From the point of view of the tool, a reduction-adversary is just another program fragment.

```
program f_owp := {
  x <$ uniform UNIV;
  adv_in <- f x;
  call B;
  x' <- adv_out;
}.
```

This is the game that defines that `f` is a one-way permutation (with respect to adversary `B`). Note the duplication: We had to state the definition (or more to the point, the main game from the definition) twice, once for `f`, once for `g`. This is a limitation of the tool; we do not have a convenient way of making parametric definitions. In our case study [32], where we proved quantum security of a variant of the Fujisaki-Okamoto transform [16], we ended up using some preprocessing scripts to generate multiple variants of the same files. A full-fledged module system with parametric definitions would be very useful. We stress, however, that this also leads to a considerably more complex background theory. For example, in the EasyCrypt tool, there are cloned theories and program modules that are quite powerful, but their formal meaning is unspecified at this point, and it is not currently clear how to define them in a way that existing tactics work in a sound way.[9]

[9] We have no reference for this, but this is our conclusion after discussions with the authors.

```
ambient var rho : program_state.
lemma security: Pr[x=x' : g_owp(rho)] = Pr[x=x' : f_owp(rho)].
```

Here we state the final result. This needs to be read as: "for the same initial state ρ, the probability that $x = x'$ is the same in programs `g_owp` and `f_owp`". Note that we don't show the actual security claim "if `f` is a (τ, ϵ)-one-way permutation, then `g` is a $(\tau - \tau_f, \epsilon)$-one-way permutation". Instead, we exhibit an *explicit* reduction adversary `B`, and the actual security claim is now a "trivial" corollary from:

- the fact that `lemma security` holds for all `A`
- the "obvious" fact that the runtime of `B` is that of `A` plus τ_f. (Recall: τ_f is the time for evaluating `f`.)

Now relying on this "obvious" fact is a big limitation. We cannot close this gap with `qrhl-tool` because it does not allow us to reason about runtime. This is a common phenomenon in crypto theorem provers. In EasyCrypt, there are extensions for reasoning about runtime, but the final word about how best to do this seems to be still out.

Internally, this command reads the statement of the lemma using the Isabelle parser, and creates an Isabelle goal internally. The goal is shown to the user (through the Isabelle pretty printer). On the Isabelle side, not much happens either. All that is done in the background theories (`QRHL.QRHL`) is to axiomatize a constant `probability` with custom syntax `Pr[...]`, but without giving it an actual definition. (Again, this gives us a migration path to foundationality.)

```
  byqrhl.
```

Here it gets interesting. `byqrhl` is a tactic that transforms a subgoal of the form `Pr[...] = Pr[...]` (or with $\leq$ or $\geq$) into a qRHL judgment that is sufficient for proving this goal. The resulting judgment is represented internally as a quadruple of pre-/postcondition and two programs. Pre- and postcondition are terms fully represented in Isabelle; the programs are represented in qrhl-tool. Qrhl-tool prints the current goal as:

```
Pre:   𝔠𝔩𝔞[x1 = x2 ∧ adv_in1 = adv_in2 ∧ cglobA1 = cglobA2
       ∧ adv_out1 = adv_out2 ∧ x'1 = x'2] ⊓ ⟦qglobA1⟧ ≡q ⟦qglobA2⟧

Left:  (1) call g_owp;
Right: (1) call f_owp;

Post:  𝔠𝔩𝔞[(x1 = x'1) = (x2 = x'2)]
```

While `byqrhl` is simply an axiomatized tactic (meaning, we have hardcoded in the qrhl-tool source code what it does), the representation of the pre-/postcondition in Isabelle is fully defined, and all lemmas we use when working with these are fully formalized! (The programs are simply kept in internal representation without special Isabelle support.) In a nutshell, the pre-/postconditions are of type `mem2 ell2 ccsubspace`, where `mem2` is the type of pairs of memories ($\mathcal{M} \times \mathcal{M}$ in our notation from Section 3). Then `mem2 ell2` is the Hilbert space with states $|(m_1, m_2)\rangle$ for $m_1, m_2 \in \mathcal{M}$ as a basis. And `mem2 ell2 ccsubspace` is the type of all (topologically closed) subspaces of `mem2 ell2`. Comparing this with our discussion from Section 3, we see that this is exactly the type we wanted for representing quantum predicates on pairs of memories. However, since we will need to reason in Isabelle about these predicates (in a foundational way!), these types need to be fully defined. And since `mem2` is infinite, this needs the full machinery of infinite dimensional Hilbert spaces (and operators thereon, and subspaces thereof, etc.). In fact, the development of this background math has

given rise to several large libraries of Isabelle formalization [14, 36, 35]. This background material is probably the largest effort in developing the theorem prover. (The hardcoded tactics are relatively simple Scala methods instead.)

To explain what exactly is happening in the postcondition: We see a classical expression `(x1 = x'1) = (x2 = x'2)`, i.e., an expression involving only the classical variables of the programs. Intuitively, this is the postcondition we need to hold for the two probabilities from *lemma security* to be equal. However, we need a quantum predicate here, i.e., a subspace, not a boolean. For this, we have simply defined an Isabelle function `𝔠𝔩𝔞[...]` that maps a boolean to a subspace; this allows us to embed classical predicates into quantum predicates.[10] Looking at the precondition, we additionally see ⊓, meaning the intersection of two subspaces, and ≡q, meaning quantum equality (discussed in Section 3). So with this precondition, the qRHL judgment essentially states that the postcondition needs to follow if the initial states of the two programs satisfy that all variables (including the quantum variables) have the same values.

The proof continues:

```
inline g_owp.
inline f_owp.
inline B.
wp 1 2.
equal 1.
 1: { simp. }
wp 1 1.
rnd x,x <- map_distr (λx. (x, f x)) (uniform UNIV).
skip.
```

A sequence of tactics, inlining the definitions of the programs we defined, and applying various qRHL reasoning rules. (We omit the details here, see the manual [26] instead.) The last tactic `skip` transforms the qRHL subgoal (which now has two empty programs) into an inequality between subspaces. (Specifically, the precondition at that point needs to be a subspace of the postcondition.) We get the goal:

```
𝔠𝔩𝔞[x1 = x2 ∧ adv_in1 = adv_in2 ∧ cglobA1 = cglobA2
∧ adv_out1 = adv_out2 ∧ x'1 = x'2] ⊓ ⟦qglobA1⟧ ≡q ⟦qglobA2⟧
≤ 𝔠𝔩𝔞[map_distr fst (map_distr (λx. (x, f x)) (uniform UNIV))
= uniform UNIV ∧ map_distr snd (map_distr (λx. (x, f x))
(uniform UNIV)) = uniform UNIV] ⊓ (⨅z∈supp (map_distr (λx. (x, f x))
(uniform UNIV)). (⨅adv_out2 adv_out1. 𝔠𝔩𝔞[adv_out1 ≠ adv_out2]
+ 𝔠𝔩𝔞[(fst z = adv_out1) = (snd z = f adv_out2)])
⊓ 𝔠𝔩𝔞[cglobA1 = cglobA2 ∧ g (fst z) = f (snd z)
∧ adv_out1 = adv_out2] ⊓ ⟦qglobA1⟧ ≡q ⟦qglobA2⟧)
```

Admittedly near impossible to read. But fortunately the next tactic manages to just prove it:

```
simp! g_def bij_f.
```

This runs the Isabelle simplifier, and due to the rich math setup, it solves the goal immediately. The proof then ends with:

```
qed.
```

[10] Formally, `𝔠𝔩𝔞[False]` is defined as the 0-subspace, and `𝔠𝔩𝔞[True]` as the whole subspace.

Note that we did the whole game-based proof in one step. We could have followed the game sequence from the proof of Theorem 2 and made one `Pr[...]=Pr[...]` lemma for each step. This would have been just as easy, but the proof here was simple enough that qRHL can do it in one go.

6 Future work

In our opinion, the biggest limitation of the existing qrhl-tool is the treatment of local variables and the lack of procedure parameters and return values. As discussed in Section 3, these are not straightforward to handle in the quantum setting. We are currently developing a large refactoring of qrhl-tool [25] that has a systematic and logically clean treatment of these. To model quantum variables (and to be able to pass them around nicely to subprocedures), we use the formalism of "quantum references" [34]. These are quantum analogues to lenses in functional programming, i.e., they can point into part of a larger data structure. We use these to let quantum variables be pointers into some larger quantum memory. We already have extensive formalization of references in Isabelle/HOL [33, 27], but lots still needs to be done. The refactoring is currently in an unusable state.

An alternative direction is a complete rewrite inside a theorem prover, e.g., Lean (abandoning the "hybrid architecture"). We have not yet decided whether this is a suitable direction to take, but given the recent surge in popularity of Lean, it might be worthwhile.

References

1 David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000. doi:10.1007/3-540-46419-0_3.

2 Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. EasyPQC: Verifying post-quantum cryptography. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2564–2586. ACM, 2021. doi:10.1145/3460120.3484567.

3 Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII, FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2014. doi:10.1007/978-3-319-10082-1_6.

4 Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009. doi:10.1145/1480881.1480894.

5 Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011. doi:10.1007/978-3-642-22792-9_5.

6 David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *Journal of Cryptology*, 33:494–566, 2020. doi:10.1007/s00145-019-09341-z.

7 Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security, CCS 1993*, pages 62–73. ACM, 1993. doi:10.1145/168588.168596.

8 Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006. doi:10.1007/11761679_25.

9 Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*, pages 14–25. ACM, 2004. doi:10.1145/964001.964003.

10 Garrett Birkhoff and John von Neumann. The logic of quantum mechanics. *Annals of Mathematics*, 37(4):823–843, 1936. doi:10.2307/1968621.

11 Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008. doi:10.1109/TDSC.2007.1005.

12 Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016. doi:10.6092/issn.1972-5787/4593.

13 Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 41–69. Springer, 2011. doi:10.1007/978-3-642-25385-0_3.

14 José Manuel Rodríguez Caballero and Dominique Unruh. Complex bounded operators. *Archive of Formal Proofs*, September 2021. Formal proof development. URL: https://www.isa-afp.org/entries/Complex_Bounded_Operators.html.

15 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

16 Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26:80–101, 2013. doi:10.1007/s00145-011-9114-1.

17 Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. *ACM Trans. Program. Lang. Syst.*, 45(3):1–61, 2023. doi:10.1145/3594735.

18 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.

19 Microsoft. Language server protocol specification. https://microsoft.github.io/language-server-protocol/.

20 Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *Principles of Security and Trust – POST 2015*, volume 9036 of *Lecture Notes in Computer Science*, pages 53–72. Springer, 2015. doi:10.1007/978-3-662-46666-7_4.

21 Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Paper 2004/332, 2004. URL: https://eprint.iacr.org/2004/332.

22 Dominique Unruh. hashedcomputation. Subdirectory `hashedcomputation` of the qrhl-tool repository, https://github.com/dominique-unruh/qrhl-tool/tree/master/hashedcomputation.

23 Dominique Unruh. One-way permutation example for qrhl-tool. Files `examples/OWP.thy` and `examples/owp.qrhl` in the qrhl-tool repository at commit `328ad96`. URL: https://github.com/dominique-unruh/qrhl-tool/tree/328ad96e726e087dd1c22f0a94b3b333b3384ebd/examples/.

24 Dominique Unruh. qrhl-tool. Source code at https://github.com/dominique-unruh/qrhl-tool. URL: https://dominique-unruh.github.io/qrhl-tool/.

25 Dominique Unruh. qrhl-tool, branch `topic-registers`. URL: https://github.com/dominique-unruh/qrhl-tool/tree/topic-registers.

26 Dominique Unruh. qrhl-tool manual, version 0.7.5. Also included in the qrhl-tool binary distribution. URL: https://dominique-unruh.github.io/qrhl-tool/manual-0.7.5.pdf.

27 Dominique Unruh. Registers2 Isabelle formalization. Subdirectory `isabelle-thys/Registers2` of the qrhl-tool repository (branch `topic-registers`), `https://github.com/dominique-unruh/qrhl-tool/tree/topic-registers/isabelle-thys/Registers2`.

28 Dominique Unruh. scala-isabelle. `https://github.com/dominique-unruh/scala-isabelle`.

29 Dominique Unruh. Quantum proofs of knowledge. In *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2012. `doi:10.1007/978-3-642-29011-4_10`.

30 Dominique Unruh. Quantum relational Hoare logic. *Proceedings of the ACM on Programming Languages*, 3(POPL):33:1–31, 2019. `doi:10.1145/3290346`.

31 Dominique Unruh. Local variables and quantum relational Hoare logic, 2020. `arXiv:2007.14155`.

32 Dominique Unruh. Post-quantum verification of Fujisaki-Okamoto. In *Advances in Cryptology – ASIACRYPT 2020*, volume 12491 of *Lecture Notes in Computer Science*, pages 321–352. Springer, 2020. `doi:10.1007/978-3-030-64837-4_11`.

33 Dominique Unruh. Quantum and classical registers. *Archive of Formal Proofs*, October 2021. Formal proof development. URL: `https://www.isa-afp.org/entries/Registers.html`.

34 Dominique Unruh. Quantum references, 2021. `arXiv:2105.10914`.

35 Dominique Unruh. The tensor product on Hilbert spaces. *Archive of Formal Proofs*, September 2024. Formal proof development. URL: `https://www.isa-afp.org/entries/Hilbert_Space_Tensor_Product.html`.

36 Dominique Unruh and José Manuel Rodríguez Caballero. Complex bounded operators in Isabelle/HOL. In *Seventeenth Conference on Interactive Theorem Proving – ITP 2026*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. To appear. Preprint/full version is arXiv:2512.05878 [cs.LO]. `doi:10.48550/arXiv.2512.05878`.

37 John Watrous. Zero-knowledge against quantum attacks. *SIAM Journal on Computing*, 39(1):25–58, 2009. `doi:10.1137/060670997`.