A Quickstart Guide to using the qRHL-tool

Tejas Anil Shah (tejas.anil.shah@ut.ee)
Supervised by: Dominique Unruh (unruh@ut.ee)

Table of Contents

Formal Verification: A Primer

All proofs in written LaTeX are correct, until proven otherwise.

Although a joke there is a kernel of truth in it. The point is that mathematical proofs are hard for humans to read, and especially proofs written by other humans. Sometimes theorems/proofs which have logical flaws are published even after being peer-reviewed, meaning errors go undetected even after reviews. In the general setting, authors can publish corrections, or the papers can be retracted if the flaws are too large to be corrected. But in contexts where the results from these claims are used in industries and products, the cost of unverified results can be too high. For instance, it was shown that a standard proof of the PRF/PRP switching lemma, which used in textbooks as well, was flawed. When it comes to cryptographic proofs and claims made by protocols there is a need for even more rigorous verification. That's where computer aided/assisted verification comes in.

The idea behind computer assisted formal proofs is to first define a set of axioms that we generally regard to be true. Then we use these axioms as tactics to be able to prove complex statements. The only difference is that since the computer can only understand statements that we define to be true, it can step through proofs step by step and verify the claims that we make. Whenever we try to make claims that don't follow from the axioms or results that we prove using the axioms the computer will not be able to verify the correctness of the claim.

In essence we define valid rules for theorem proving and in a way tell the computer that only things that play by these rules are allowed. If any claim that we make doesn't follow, then it is unverified by default. This shifts the burden of the proof on the person making the claim. The verifiers simply need to run the proof on a computer and see if the computer verifies it. An additional requirement is to check the statement and definitions are indeed what is expected as well. If the formal logic backing the verification is sound, then it becomes impossible to make false claims and verify them. With verified proofs we solve the problem of incorrect proofs.

Formal Verification in Cryptography and Quantum Cryptography

Modern cryptography is based on various mathematical functions that are easy to compute one way and extremely hard to reverse. This implies that the security of a protocol depends on quantifying the difficulty of decrypting ciphertexts using different methods. One way to quantify the security of an encryption protocol is by measuring an adversary's advantage. The idea behind building secure protocols ends up building protocols that offer very little advantage to adversaries. So if cryptographers invent a protocol, they also make claims about how secure it is. They do so by making mathematical claims about the adversary's advantage or other properties of the protocol, generally with hand-written or LaTeX proofs. This is where computer aided verification gives researchers the ability to verify the security claims being made about protocols. There already are specialized tools which help with formal verification of cryptographic protocols like EasyCrypt, but they lack the logic and techniques that we require to work with Post Quantum Crypto.

With practical quantum computers being built, we are aware that they pose a threat to classical cryptography. Although work is being done to make classical protocols quantum-safe and new protocols are being invented with claims of quantum safety, there is a need for us to be able to verify the claims of quantum safety. This is the use case of the qRHL-tool. It is based on Quantum Relational Hoare Logic (qRHL), and can be used to write proofs for protocols that use ideas that are based on Quantum Information.

A point to note about both qRHL-tool and EasyCrypt is that not all steps are broken down to the elementary rules. All the tactics and simplification rules are axiomatized and are backed with proof in the paper. The criticism for this could be that it falls prey to the problem of LaTeX based proofs that it set out to solve. The answer to that would be that if the rules of qRHL are sound even to a certain extent the tool allows us to verify

security protocols which can be put to use. A protocol verified by an imperfect tool is a huge improvement over a protocol with a handwritten proof.

Intended Audience

This guide is for people who have some background in the theory of Quantum Computing and are beginning their journey into the realms of formal proofs and verification for Quantum Cryptography.

A strong background in mathematics is recommended.
To be more specific it requires an understanding of Group theory, Probability theory, and Linear Algebra.
Some exposure to Cryptography is also needed, specifically having some exposure to game based security proofs and ideas is recommended.

If you are a complete beginner, this guide will not be very helpful.
We recommend building a strong foundation of Quantum Information Theory and then returning to this tool.
A good place to start would be to take the Quantum Cryptography course offered by Prof. Unruh. The lectures are open access, and these lectures build the required theory with the only prerequisite of enjoying math and curiosity to learn.

Additionally, some students have also benefited from the following resources as well:

1. Essence of Linear Algebra:
   This is a series of bite sized lectures on YouTube by 3Blue1Brown, which help you develop an intuitive understanding of the concepts and ideas of Linear Algebra. This has helped some students a lot.
   It is highly recommended to watch these to refresh your concepts of linear algebra.
2. IBM's Qiskit Summer School of 2020:
   Another series of Youtube videos by IBM's qiskit team. This is a good place to start if you are a complete beginner at Quantum Information Theory and are just getting started.
3. Quantum Country:
   If you aren't a visual learner and prefer reading texts instead, Quantum Country is an interactive web-based textbook which has active recall and spaced repetition built in. It is a great way to make concepts stick.
4. Quantum Computation and Quantum Information:
   If you prefer something completely old school, and want to do things your own way, nothing is better than this classic textbook by Isaac Chuang and Michael Nielsen.

Architecture

The qRHL-tool relies on three main components.

1. The core qRHL-tool: Written in Scala, this is the main component which has the logic and tactics that are described in the paper.
2. Isabelle/HOL Backend: Isabelle is a generic proof assistant which does the heavy lifting for us when it comes to working with ambient logic (statements which don't involve qRHL). Isabelle also comes with it's own frontend, that we might use from time-to-time.
3. Proof General Frontend: A generic interface for theorem provers based on Emacs. The interactive nature of Proof General makes it a great choice for writing formal proofs.

Typically writing proofs with the tool will involve a workflow that looks something like this:

1. Creating .thy (theory) files for Isabelle which house, like name suggests, the "theory" or definitions that we require for our protocols. Any supporting lemmas that don't need qRHL tactics can also be written in the .thy files
2. Creating .qrhl files which will have the main program/proof. We connect the qrhl files with isabelle theory files and backend by using "isabelle filename." (where filename.thy is the theory file that we want the qrhl file to use.) We will generally work on .qrhl files in Proof General.

Setup

Refer to the web-page of the qRHL-tool for up-to-date instructions about installation and set-up.

Emacs/Proof General Crash Course

Since ProofGeneral is based on Emacs, some familiarity with it is required.
If you have worked with Emacs and Proof General before you can safely skip this section.

To put it simply, Emacs is a powerful, and extensible text-editor. It allows you to do a lot of things from the keyboard, and this is the main idea behind it as well. You shouldn't need to reach for the mouse to be able to do things. The interface doesn't give any clues about what it is capable of and this ends up being confusing to most beginners. So in order to prevent that from happening here is a crash course that will help you get started with Emacs/ Proof General.

The keyboard is your new best friend. Emacs commands are invoked using a pattern of keyboard shortcuts, and the commands revolve around two keys.

1. The CONTROL key, generally abbreviated as 'C', is the Ctrl key of the keyboard.
2. The META key, generally abbreviated as 'M', is generally the Alt key of the keyboard.

So keybindings which read "C-x" imply that you need to press the CTRL key and then the x key. Similar to the shortcut Ctrl-a to select everything in most text editors. Generally, the commands come in pairs of these keys. For example: Exiting the editor is "C-x C-c", this implies that you first need to press Ctrl-x, and let go of the keys, and then press Ctrl-c. This is how most common commands work in Emacs.

To get started we don't need to know everything that Emacs is capable of.
The most important commands that we need at the beginning are:

Action	Key Binding
Visit a file	C-x C-f
Evaluate Next Line in a proof	C-c C-n
Evaluate up to the cursor position in a proof	C-c C-RET
Undo evaluation in a proof	C-c C-u
Save file	C-x C-s
Exit	C-x C-c

Note: The Emacs guided tour is very helpful as well. The editor gives you the option of taking the tour on the first launch of the application. Additionally, there are plenty of video tutorials on Youtube that are helpful in breaking down Emacs.

Isabelle

Sometimes we might need to use Isabelle's frontend to edit the .thy (theory) files.
Use the "run-isabelle{.sh/.bat}" command to start Isabelle.
Isabelle has a slightly more advanced GUI than Emacs.
Finding your way around to opening and editing files shouldn't be a problem.

Get your hands dirty

Please make sure you have set up the qRHL tool before going further. Most of the explanation of the code is as comments in the examples. This section explains the big picture and the reasoning of the proofs. It is better to see it in action in code.

In this guide, we will go over one example in detail and there are more examples that are bundled with the tool itself in the "*examples*" folder.

Security Analysis of the classical One Time Pad (OTP)

One of the first security schemes that all textbooks and courses go over is the OTP. The scheme works as follows. You draw a shared key which is the same length as the message that you want to encrypt. Encryption is the XOR operation of the message and the key, and since XOR is its own inverse, decryption is the XOR operation of the ciphertext and the key. This can be expressed as:

$$k = \{0,1\}^n$$
$$m = \{0,1\}^n$$

$$E(k,m) = k \oplus m$$

$$D(k,c) = k \oplus c$$

One way of proving the security of the OTP is to use a game based proof. In this example we will prove the notion of Real Or Random - One Time - Chosen Plaintext Attack of the classical OTP. The main idea is that any given adversary cannot tell the difference between a random value, and the output of the OTP. In essence, it boils down to proving that applying a bijective function (XOR) on a randomly chosen value (the key) yields an output which is indistinguishable from randomness. Although strictly speaking still in the classical domain, we chose this example since it demonstrates many of the working principles and tactics in action.

Now please run "proofgeneral{.sh, .bat}" to start Proof General, and visit the file OTP.qrhl.
Again this file can be found in the examples folder of the qrhl-tool.

See the appendix to see the code with some syntax highlighting. (Emacs is completely plain text, so it is quite hard to separate the code from the comments visually.)

Conclusion

Hopefully, this quick-start guide has helped you get started with working on the qrhl-tool. Admittedly, the underlying material and logic requires more examples, stepping stones, and exercises for students/researchers to be able to work towards using the full power of the qrhl-tool.
This guide only marks the beginning of attempts to do so.

Appendix

We will include the code with some syntax highlighting that might help in readability of the code in this guide.

OTP.thy

```
theory OTP
  imports QRHL.QRHL
begin

declare_variable_type msg :: <{finite, xor_group}>

type_synonym key = msg
type_synonym ciph = msg

definition otp :: <key ⇒ msg ⇒ ciph> where
  <otp k m = k + m>

end
```

OTP.qrhl

```
# Please read the accompanying quickstart guide to understand
# the high-level proof sketch, then return to this file to step through
# and get an understanding of the tactics

# The quickstart guide is available on the webpage.
# https://dominique-unruh.github.io/qrhl-tool/quickstart_guide-qRHL-tool.pdf
# Documentation: Tejas Anil Shah
# Supervision: Dominique Unruh

# Once you open this file with Proof General
# You should be in an Emacs environment
# There are three panes in the window.
# 1. Script
# 2. Goals
# 3. Response
# You can execute the code line by line inorder to interactively
# prove a mathematical statement.
# Code execution is as follows:
# Process one step: C-c C-n (CTRL-c CTRL-n)
# Backtrack one step: C-c C-u
# Process upto the cursor position: C-c C-RET (RET is generally the ENTER key)
# Use the above command to skip through the comments and jump to the executable code.

# The qrhl-tool uses Isabelle/HOL as a backend, so the first step is
# to initialize Isabelle as follows:

isabelle OTP.
# This line initializes isabelle, and
```

```
# uses the OTP.thy file.
# The OTP.thy only contains a few type definitions
# and the definition of the encryption (XOR operation)
# You can take a quick glance at it and return to this file
# You can do so by using the Isabelle editor
# Use "run-isabelle{.sh/.bat}" script
# and then opening the OTP.thy file with it

# We first declare variables that we will use
# An ambient variable is an arbitrary but fixed value
# Meaning once an ambient variable is picked to be a certain value
# it can't be changed

ambient var m0 : msg.
ambient var rho : program_state.

# Similarly we define standalone variables as well
# The classical variables are declared using the syntax
# classical var variable_name : variable_type

classical var c : ciph.
classical var k : key.

# In the OTP.thy file, we have defined the types ciph, key to be type_synonyms of the
"msg" variable_type
# This makes our intent to use these variables clear. Although under the hood they
resolve to the same variable type.

classical var b : bit.

# The bit variable_type is isomorphic to a boolean variable type

# In order to account for the internal state of the adversary
# we define two variables (One for the classical state and one for the quantum state of
the adversary)
classical var cglobA : infinite.
quantum var qglobA : infinite.

# Now we define the adversary as follows
adversary A free c, b, cglobA, qglobA.

# This allows the adversary A to access the listed free variables
# c - intended for input
# b - intended for output
# cglobA, qgbloA - internal state of adversary
# This implies that the adversary A can read the ciphertext c, and output b according
the
# it's internal state

# With all the variables and states defined
# We define the enc_fixed, in which the adversary gets
# the encryption of m0.
```

```
# Since there are no restrictions on m0, this covers the case where m0 is a completely
arbitrary message.
# Intuitively, we can think of it as chosen by the adversary.

# After encryption we call the adversary.
program enc_fixed := {
  k <$ uniform UNIV;
  c <- otp k m0;
  call A;
}.

# In our definition of the adversary A, we made sure that it has access to
# the ciphertext, c, and the bit, b. So "call A" intuitively means: b <- A(c)

# Similarly we define the program where the adversary is given
# something random
program random := {
  c <$ uniform UNIV;
  call A;
}.

# So far we have defined the programs enc_fixed and random.
# "enc_fixed" picks a uniformly random key, k
# Encrypts m0 with k and stores the ciphertext to c
# Where as, "random" simply picks something random and passes it off as the ciphertext
# In both cases the adversary has access to c

# We know that OTP has perfect secrecy.
# This means that an adversary cannot distinguish between a real encryption
# of a message and a completely random message.
# We will shortly make a claim that supports this idea.

# We're through with the setup
# Tools like Isabelle and qrhl-tool allow us to:
# 1. State a claim
# 2. Use tactics which could be axioms or other provable results to prove what we
stated

# In this lemma, we claim that
# The probability that the adversary outputs 1 when given an actual encryption
# is equal to the probability that it outputs 1 when given some random input

lemma secure: Pr[b=1 : enc_fixed(rho)] = Pr[b=1: random(rho)].

# Up until this point the Goals pane was empty.
# Once we say that we have a lemma to prove. The Goals pane is populated with
# what needs to be proven, in this case a lemma called "secure".

# To instruct the qrhl-tool to use qRHL, we use the tactic "byqrhl"
# byqrhl transforms a goal of the form:
# Pr [e: P(rho)] = Pr[e': P'(rho)]; P and P' are the programs
# to something that we can attack with qRHL tactics
```

```
byqrhl.

# At this point the goal is stated long form
# The long form is equivalent to {pre} left ~ right {post}
# The long form is simply more readable when there are more variables and conditions

# The way to read the Goal here is to think that
# If the "Pre" conditions hold, and we run the programs ("Left", and "Right")
# our goal is to prove that the "Post" condition holds as well
# There is a caveat here about what running both the programs together means.
# At this point it doesn't matter too much, but if you would like to delve further into
it
# please refer to the qRHL paper

# The variables with the "1" appended to the end are accessed by the "Left" program
# Similarly, the variable appended by "2" are accessed by the "Right" program
# 𝔠𝔩 refers to the classical variables that we have
# The symbol ≡𝔮 is used to talk about quantum equality
# which refers to the notion of equality in the quantum setting
# The symbol ⊓ is to be interpreted as an "and" for quantum predicates.
# Formally this implies the intersection of the subspaces

# In essence "Pre" states that we start with the fact that the initial states of the
memories
# (both classical and quantum) that the left and right programs work on are equal.
# Where as the "Post" condition states that "b1 = b2" since they are both boolean
variables and equal to 1
# Our objective is to work towards showing that this predicate holds.


# Those familiar with EasyCrypt might find this intuitive and familiar.
# Again a quick summary is that
# 1. If we start with same values in both the program (Pre)
# 2. Run programs "enc_fixed" and "random" (Left and Right)
# 3. They both return the value same value b (Post)

# To get started here, we need to be able
# to open up the programs: enc_fixed and random
# To do so we use the tactic "inline"
# It simply replaces the programs with their definitions
inline enc_fixed.
inline random.

# When working with formal verification a general strategy is to work towards the goal
# starting from the end of the programs and working our way backwards.

# Now if we observe our goal state, in both the left program, and the right program
# we have an equal instruction, in situations like these we can use
# the tactic "equal" to remove same statements from the end subject to some conditions.
# Since we have the same instruction of "call A" on both sides, we can use the tactic
here.
equal.
```

```
# Notice that now we have 2 sub-goals
# Many a times after using the equal tactic some of these sub-goals can be
# dispensed off by using the tactic "simp"
# This invokes the Isabelle/HOL simplifier on the current goal
# In our case we can observe that the sub-goal is true
# since two classical variables can either be equal or not equal
# So applying the tactic "simp" dispenses the subgoal.

# qRHL-tool also supports braced sub-goal proving
# Opening a brace will clear out the other goals when the scope is in the brace.
# Decluttering the workspace.
{
  simp.
}

# The post-condition consists of a term c1 = c2
# c1 in our case is equal to the OTP encryption of
# a  k (random key) and some m0
# In order for us to make progress we need to be able to replace c1
# and then we can try to reason about the distributions of k and c
# So we use the tactic wp side (where side = left,right)
# It removes the last statement from the target side
# In our case we use wp left to get rid of the last line
# from the left program

wp left.

# Observe that the tactic replaced c1 in the post-condition with otp k1 m0
# wp can also be used as follows:
# wp side x (Simply applying wp side x times)
# Ex: wp left 2 (wp left. wp left.)
#     wp n m (wp left n. wp right m.)


# If we look at the goal now, we see that otp k1 m0 = c2 occurs in the postcondition
# This is what we need to work towards
# This implies that we need to draw k1, and c in a way which
# allows us to link them, but when we look at them independently they
# still come from uniform distributions
# The tactic that allows us to do so is the rnd tactic

rnd k,c <- map_distr (%k. (k, otp k m0)) (uniform UNIV :: key distr).
simp.

# Here rnd k,c tactic picks the values in a specific way.
# 1. The type is uniform UNIV :: key distr
#    They drawn from a uniform distribution from all values (UNIV) since they are keys
# 2. map_distr (%k. (k, otp k m0)) maps k to (k, otp k m0)
#    When applied to uniform UNIV, we get a distribution of pairs of k,c
#    where the first component is uniformly chosen k,
#    and the second is the otp k m0 using that value k
```

```
# This results in a distribution which always gives pairs (k,c) that hold the
relationship c = otp k m0

# So the tactic rnd k,c <- ... essentially picks k,c together according to the chosen
distribution
# The requirement that will be imposed in the new post-condition is that these
distributions of k and c
# should separately be like the sampling in the left/right programs.

#  - tactic rnd k,c <- some-distr then picks k,c "together" according to the
distribution some-distr
#  - only requirement (will be checked in the new postcondition): the distributions of
k and c (separately) are like the sampling in the left/right program


# Is is encouraged to comment out the tactics "rnd k,c ..." and "simp" and
# run the bad approach described below to understand why it is a bad approach
# Bad approach 1: (treats k,c as arbitrary unrelated values)
# wp 1 1.
# simp.
# Taking this bad approach brings us to the state in the post-condition
# which says that ∏x xa | otp xa m0 = x which reads that
# For all x and xa, otp xa m0 = x, this clearly can't happen for all x and xa.

# Another bad approach is to simply run the rnd without specifying how the values are
drawn
# Bad approach 2: (samples k,c jointly, but with k=c)
# rnd.
# simp.

# This approach brings us to a post-condition part of which reads ∏x | otp x m0 = x
# Which again can't happen for all x.

# At this point please make sure both the bad approaches are commented out
# And the correct approach with "rnd k,c ..." and "simp" are not commented

# Now the goal state has only skip as programs in both
# the left side and the right side.
# Inorder to move away from this we apply the tactic skip
# It converts a qRHL subgoal of the form {A} skip ~ skip {B} into
# an ambient logic subgoal
skip.

simp.

# Now note the goal
# It says "map_distr (λx. otp x m0) (uniform UNIV) ≠ uniform UNIV implies something".
# "simp" did not realize that in fact "map_distr (λx. otp x m0) (uniform UNIV) =
uniform UNIV" holds
# and that would imply that the goal would be trivial.
# Since we would have a goal state in which "uniform UNIV ≠ uniform UNIV" (which is
clearly false) holds
```

```
# If a false condition holds any implication can be derived from it
# So the fix (in this case) is to tell "simp" the definition of otp. Like so:

simp otp_def.

# We could've already given the definition of otp_def to simp right from the beginning
# basically combining the previous two simp tactics into just one.
# You can comment out the previous simp and run only simp otp_def and that would also
be enough.
# We have them split into two to show the intermediate steps.

# In complex cases, we may need to prove helper lemmas
# For instance we could have created a separate lemma like so
# lemma map_univ: map_distr (λx. otp x m0) (uniform UNIV) = uniform UNIV.
# Proved the lemma, and used simp with the lemma to make progress.
# But in this case we were lucky and simp was strong enough to prove the goal.

# Once there are no goals to be proved we can end the proof with qed.
qed.
```