Tutorial: High-Level Cryptol Verification with the Isabelle Theorem Prover

Joe Hurd
joe@galois.com
Galois, Inc.

January 20, 2012

Contents

Chapter 1

Introduction

CHAPTER GOALS:

- Describe the situations in which it is useful to perform high-level Cryptol verification with the Isabelle theorem prover.
- State the purpose and assumptions of this tutorial.
- Set up the environment required for verifying Cryptol programs using the Isabelle theorem prover.

A critical step in building a system in which we can place a high degree of trust is showing that it implements precisely what it is specified to do. The process by which we show that the implementation matches the specification has an impact on the level of trust we can place in the system. An untested piece of software that compiles correctly generally would be regarded as less trusted than a similar program that is accompanied by a small test suite that it must pass. As the test suite grows, we learn more about how a program behaves under different situations. Unfortunately, there are limits to brute force testing—we can only plausibly work with test suites that are limited in size due to the computational cost of performing them. For a complex program, even a relatively large test suite may cover only a tiny fraction of the set of possible inputs that are possible. A better way must be taken if we want to elevate our confidence and trust in a system beyond that which we can achieve by brute force testing alone. For this, we turn to formal verification.

In this document, we focus on the specific instance of this problem focused on verification of programs specified in the Cryptol language. The Cryptol to Isabelle translator allows us to convert Cryptol programs into Isabelle theories, which can be read into the Isabelle theorem prover and proved to satisfy a logical specification. The logical specification can be directly expressed in terms of a mathematical theory that has been formalized using the Isabelle theorem prover, or can be expressed in terms of other Cryptol programs that have been converted to Isabelle theories. Thus the two main use cases of this verification flow are as follows:

1. Implement two versions of a cryptographic operation as Cryptol programs: a reference version that emphasizes clarity of purpose; and an optimized version that emphasizes high performance of execution. Translate both versions to Isabelle theories, and use the Isabelle theorem prover to prove their equivalence.

2. Implement one version of a cryptographic operation as a Cryptol program, and translate it to an Isabelle theory. Use the Isabelle theorem prover to prove that the cryptographic operation satisfies desirable properties (e.g., an encryption operation followed by the corresponding decryption operation yields the original plaintext message).

We will illustrate both of these use cases in this tutorial.

Before we go into the details of how to set up this verification flow, it is important to cover the main strengths and weaknesses of using the Isabelle approach to verify Cryptol programs. The main strength of the approach derives from the higher order logic implemented by Isabelle, which is expressive enough to capture virtually any property of a Cryptol program's functional behavior. However, the Cryptol to Isabelle translator will only succeed on a subset of Cryptol programs. In particular higher order logic is a logic of *total* functions, so only terminating Cryptol programs can be translated to Isabelle theories. In practice this does limit the applicability of the approach, but sometimes the first step of verifying a Cryptol program using Isabelle is to formalize the termination argument so that Isabelle will admit the translated theory.

The main weakness of this verification approach is that carrying out proofs using the Isabelle theorem prover generally requires a large amount of skilled human effort. If showing the equivalence of two Cryptol programs is the goal, and it is possible to use one of the many automatic proof tools [1] that are connected to Cryptol to construct the equivalence proof, then this is usually the least effort course. However, if the two Cryptol programs are sufficiently different, perhaps because they implement two different high-level algorithms to accomplish the same task, then translating them to Isabelle and using human effort to prove their equivalence might be the only possible course to complete the verification.

1.1 About this Tutorial

The purpose of this tutorial is to teach the reader the mechanics of performing high-level Cryptol verification using the Isabelle theorem prover. We use three examples of increasing complexity to illustrate the verification flow, focusing on the new concepts that each bring to light.

The tutorial is intended for a general audience and assumes little about existing experience with software verification. We do assume that the reader has read some basic introductory materials related to the Cryptol language [4], and is sufficiently comfortable with the Isabelle theorem prover [6] to be able to read theory files, set goals and invoke tactics. The most important documents that should be read in conjunction with this tutorial include:

- Cryptol Programming Guide [2]

- An Introduction to Cryptol via Exercises
- Isabelle/HOL—A Proof Assistant for Higher-Order Logic [7]

1.2 Setting up your Environment

Before starting it is important that a working environment is available to work through the examples presented in the text. The two main tools that must be installed are Cryptol[1] and Isabelle.[2] This tutorial was prepared using Cryptol version 1.8.21—the version is printed when entering the Cryptol interpreter shell:

```
Cryptol version 1.8.21, Copyright (C) 2004-2011 Galois, Inc.
                                                www.cryptol.net
Type :? for help
Cryptol>
```

The version of Isabelle used in this tutorial is `Isabelle2009-2`.[3] Follow the standard instructions for installing both Cryptol and Isabelle.

The only other essential component of the environment is a collection of Isabelle theories that support translated Cryptol programs.[4] Here is the simple approach to installing and using the supporting theories:

1. Download the tarball `CryptolThys-2010-12-21.tar.gz` of supporting theories.[5]
2. Extract the tarball, which will create a new directory `CryptolThys`:

   ```
   % tar xvzf CryptolThys-2010-12-21.tar.gz
   ```

3. Whenever you convert a Cryptol program to an Isabelle theory file, put the theory file into the `CryptolThys` directory. When loading the theory file Isabelle will then be able to find all of the supporting files it needs.

However, this simple approach has two disadvantages: (i) Isabelle has to load all of the Cryptol background theories each time it is launched, which can take a while; and (ii) All of your theory files must live within the same directory. This means that every time we deliver a new version of the background theories, you'll have to copy all of your files into the new directory.

You can avoid both problems by editing and running the `make_cryptol_heap` script that comes with the supporting theories, to create a custom Isabelle heap image with the

[1] Available from the Cryptol Wiki at `https://www.galois.com/cryptol/wiki`
[2] Available from `http://isabelle.in.tum.de/`
[3] Available from `http://isabelle.in.tum.de/website-Isabelle2009-2/download.html`
[4] Also available from the Cryptol Wiki at `https://www.galois.com/cryptol/wiki`
[5] Also available from the Cryptol Wiki at `https://www.galois.com/cryptol/wiki`

Cryptol background theories pre-loaded. Also, you'll be able to put your own Isabelle theories wherever you want.

This is the procedure for running the `make_cryptol_heap` script.

1. Edit the `$ISABELLE_DIR` variable so that it points to the root of your local Isabelle distribution. The default setting of `ISABELLE_DIR` is designed to work for `Isabelle2009-2` on Mac OS X systems, assuming the `Isabelle2009-2` application has been placed in the `/Applications` directory.

2. Run the script from the `CryptolThys` directory:

   ```
   % ./make_cryptol_heap
   ```

This will cause Isabelle to process the background theories (you should see a lot of Isabelle messages print out), and then save the custom heap image in a place it knows about on your local machine (the exact location depends on what hardware plaform and operating system you are running).

To use the custom heap image, go to whichever directory contains your Isabelle theory files, and launch Isabelle. Within ProofGeneral, select the 'Isabelle' menu item, and then select the 'Logics' item. You should see a list of logic heap images, including 'Default', 'HOL', and 'Cryptol'. Select 'Cryptol' to use the custom Cryptol heap (you'll have to do this every time you launch Isabelle).

Chapter 2

Beginning with the Basics: A Distance Function

CHAPTER GOALS:

- Understand the key steps of the Cryptol and Isabelle workflow.
- Perform basic tests of translating Cryptol programs to Isabelle theories.
- Use Isabelle to interactively prove a simple property of a translated Cryptol program.

In this chapter we will walk through the process of translating a Cryptol program to an Isabelle theory and then using interactive proof to prove that it satisfies a particular property. For the purposes of exposition the Cryptol program and property will be very simple, and could be effectively handled by using one of the automatic proof tools connected to Cryptol. However, we will step through the procedure of translating it to an Isabelle theory and interactively proving the property, illustrating the workflow that is applied regardless of the problem size.

2.1 Translating a Cryptol Program to an Isabelle Theory

The Cryptol program we will use is a simple distance function on 32-bit words:

```
1 distance : ([32],[32]) -> [32];
2 distance(m,n) = if m <= n then n - m else m - n;
```

Assuming this is stored in the Cryptol file `distance.cry`, we can load the program into the Cryptol interpreter shell and execute it on some sample inputs:

```
Cryptol version 1.8.21, Copyright (C) 2004-2011 Galois, Inc.
                                                www.cryptol.net
Type :? for help
Cryptol> :l distance.cry
Loading "distance.cry".. Checking types.. Processing.. Done!
distance> distance(35,42)
0x00000007
distance> distance(300,45)
0x000000ff
distance>
```

Invoking the Cryptol to Isabelle translator is performed with the `:isabelle` command:

```
distance> :isabelle
(* (c) Galois, Inc. Automatically generated by Cryptol. *)

theory distance
imports Cryptol
begin

section {* Definitions *}

(* ["distance.cry", line 2, col 1] *)
(* distance : ([32],[32]) -> [32] *)
fun
   distance :: "((bvec \<times> bvec) \<Rightarrow> bvec)" where
  "distance (m, n)
     = (if m <= n
        then n - m
        else m - n)"
declare distance.simps[simp del]

end
distance>
```

By default, the `:isabelle` command translates all Cryptol functions to an Isabelle theory, and outputs the result to the shell. To redirect the output to a file, set the `outfile` Cryptol variable like so:

```
distance> :set outfile=./DistanceDef.thy
```

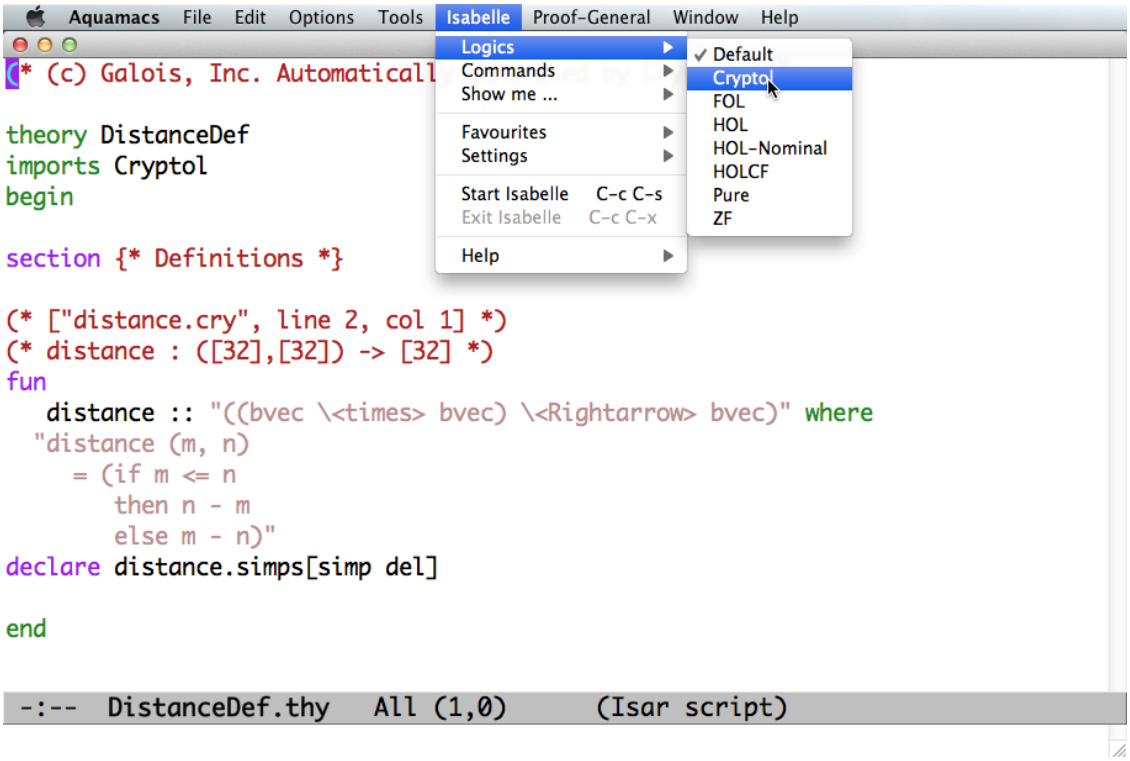

Figure 2.1: Loading the translated Cryptol `distance` function into Isabelle and selecting the 'Cryptol' heap image.

It is also possible to restrict the Cryptol functions that will be translated to an Isabelle theory. If you give the `:isabelle` command a Cryptol expression as an argument, it will translate only the Cryptol functions that are necessary to define the expression in an Isabelle theory. This is most useful by giving a tuple argument such as

```
Cryptol> :isabelle (f,g,h)
```

which will translate only the Cryptol functions `f`, `g` and `h` (together with their supporting functions) to an Isabelle theory.

At this point we have translated the Cryptol `distance` function to the Isabelle theory file `DistanceDef.thy`, and we can now launch Isabelle and load this theory. Figure 2.1 shows a screenshot of doing this, and selecting the 'Cryptol' heap image that Isabelle will use to process the theory file (as explained in Section 1.2). When this theory is processed, Isabelle will register the following definition containing the newly defined `distance` constant:

```
DistanceDef.distance.simps:
  distance (?m::bvec, ?n::bvec) = (if ?m <= ?n then ?n - ?m else ?m - ?n)
```

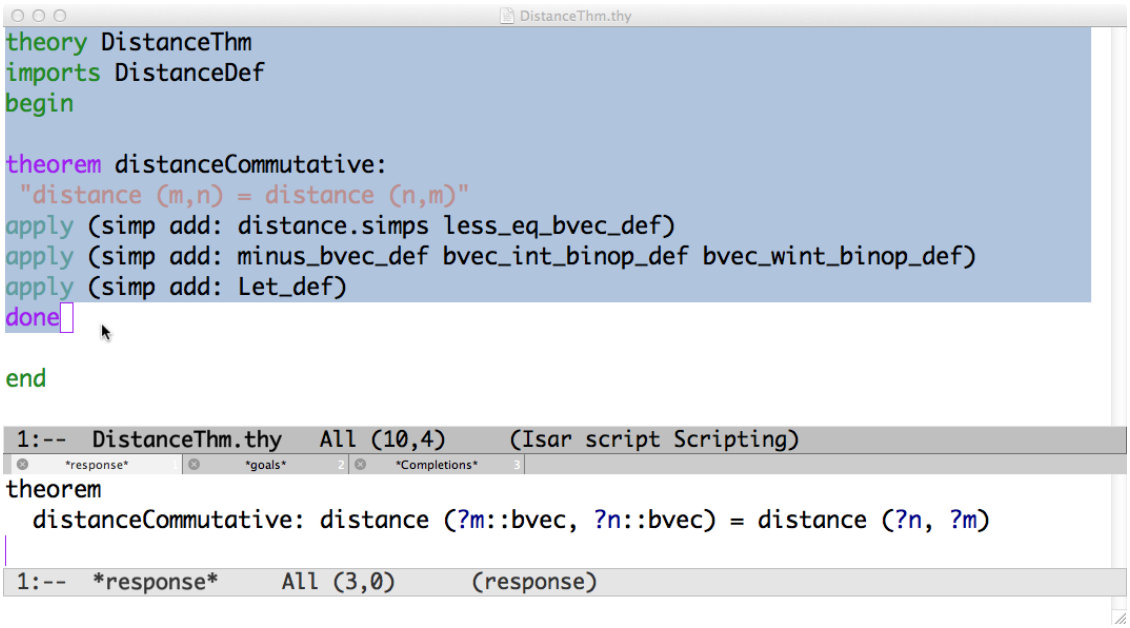

Figure 2.2: Using Isabelle to interactively prove the commutativity of the translated Cryptol `distance` function.

The Isabelle 'Show Types' setting was enabled when printing this theorem, to reveal that the Cryptol 32-bit word type has been translated to the Isabelle `bvec` type. The `bvec` type is defined in the `Bvec.thy` theory (part of the Cryptol support theories introduced in Section 1.2) to be a simple wrapper around lists of Booleans.

2.2 Interactively Proving Properties of Translated Cryptol Programs

At this point we have an automatically generated Isabelle theory `DistanceDef` containing a translation of the `distance` Cryptol function. For this example we are not going to prove that it is equivalent to another Cryptol program, but rather we will use Isabelle to interactively prove that it satisfies the basic commutativity property:

$$\forall m, n.\ \texttt{distance}(m, n) = \texttt{distance}(n, m)$$

Figure 2.2 shows a complete Isabelle theory file that: (i) pulls in the `DistanceDef` theory defining the translation of the Cryptol `distance` function; (ii) sets up the commutativity property as a goal; and (iii) proves the goal using a sequence of applied tactics. The Isabelle `*response*` window underneath confirms that the proof was successful by showing the resulting theorem.

Note that we did not add the commutativity theorem to the `DistanceDef` theory, but rather created a new `DistanceThm` theory to house the properties of the Cryptol `distance`

function. This allows us to regenerate the `DistanceDef` Isabelle theory file whenever the Cryptol program changes without losing the record of our interactive proof. Indeed, it is often the case that many proofs will succeed even when the Cryptol program changes, and others may require only small modifications.

Another point to note is that the proof of the `distance` commutativity property involved nothing more than unfolding the definition of the `bvec` operations, followed by some simple integer inequality reasoning. Usually if Isabelle is being deployed to help a verification step succeed, it is because the step relies on deeper mathematical properties, and the interactive proofs are correspondingly more complex.

At this point, we have seen the full verification workflow that one must walk through to translate Cryptol programs to Isabelle theories and interactively prove properties of the translated programs. We will now build upon this to look at programs and properties of increasing complexity.

Chapter 3

An Efficient Finite Field Division Algorithm

CHAPTER GOALS:

- Understand how nesting in Cryptol programs is translated to Isabelle theories.
- See how to create Isabelle specifications comparing translated Cryptol programs to formalized mathematical theories.
- Learn the common Cryptol verification strategy of factoring the proof into 'equivalence' and 'functionality' pieces.

The `distance` Cryptol program studied in the previous chapter was intentionally simple to serve as a 'Hello, world' example that illustrated the steps of the verification workflow: (i) translating Cryptol programs to Isabelle theories; (ii) setting goals that capture properties of the translated programs; and (iii) using Isabelle to interactively prove these goals. Larger Cryptol programs are tackled with the exact same workflow, but complications can appear at every step. In this example we will tackle a real Cryptol function taken from a library of finite field arithmetic: an efficient procedure for computing field division.

3.1 The Source Cryptol Program

Here is the starting Cryptol program, which is an implementation of Algorithm 2.22 in *Guide to Elliptic Curve Cryptography* [3]:

```
field_div : {a} (fin a, a >= 1) => ([a],[a],[a]) -> [a];
field_div(p,x,y) = egcd(p,0,y,x)
  where {
    even : {a} (fin a, a >= 0) => [a+1] -> Bit;
    even x = ~(x @ 0);

    /* Returns x / 2 (mod p) */
    half x = if even(x) then x >> 1 else drop(1, safe_add(x, p));

    /* In the code below, a is always odd */
    egcd(a,ra,b,rb) =
      if b == 0 then
        ra
      else
        egcd(if even(b) then
               (a, ra, b >> 1, half(rb))
             else if a < b then
               (a, ra, (b - a) >> 1, half(field_sub(p, rb, ra)))
             else
               (b, rb, (a - b) >> 1, half(field_sub(p, ra, rb))));
  };
```

The specification of `field_div` is as follows: given an odd prime p and two integers $0 \leq x < p$ and $0 < y < p$, the function `field_div`(p, x, y) returns an integer $0 \leq z < p$ such that $zy \bmod p = x$. The Cryptol interpreter shell can easily test this correctness property for $p = 13$, $x = 2$ and $y = 3$:

```
field> field_div(13,2,5)
0x3
field> (field_div(13,2,5) * 5) % 13
0x2
```

We can automate this testing by using Cryptol's Quickcheck tool to randomly generate test cases for the correctness property of `field_div`, but no matter how many test cases pass we can never cover the whole input space (there are an infinite number of valid inputs). In addition, this is a real example that cannot be handled by existing automatic proof tools. We therefore conclude that it is a good candidate for using the Isabelle theorem prover to prove its correctness.

3.2 Translating Nested Functions to Isabelle Theories

We use the `:isabelle` command to translate the Cryptol program to an Isabelle theory:

```
field> :set outfile=./FieldDivDef.thy
field> :isabelle field_div
```

The generated Isabelle theory contains translations of all the supporting Cryptol functions used by `field_div` (such as `field_sub` and `safe_add`), in addition to the translation of `field_div` itself. Because the `field_div` function has a nested scope containing three local functions, the translation of `field_div` results in *three* Isabelle declarations, which we will examine in turn.

The first declaration is an *Isabelle locale* that captures the three nested function definitions:

```
(* field_div_egcd_even_half ["field.cry", lines 60, 63 & 66]
 * Lift reason: [even] Needed by already lifted binding for egcd
 *              [half] Needed by already lifted binding for egcd
 *              [egcd] Recursively defined
 *)
locale field_div_egcd_even_half =
  fixes c :: "width"
    and p :: "bvec"
  begin
    (* ["field.cry", line 60, col 5] *)
    (* even : {a} (fin a, a >= 0) => [a+1] => Bit *)
    definition
       even :: "(width => (bvec => bool))" where
      "even d x = ~(bvec_nth x (0 :w 0))"

    (* ["field.cry", line 63, col 5] *)
    (* half : [b] => [b] *)
    definition
       half :: "(bvec => bvec)" where
      "half x = (if (even (c + (-1))) x
                 then cshiftr x (1 :w (max (lg2 c) 1))
                 else bdrop_lel 1 ((safe_add c) (x, p)))"

    (* ["field.cry", line 66, col 5] *)
    (* egcd : {a} (fin a,a >= 0) => ([a+1],[b],[a+1],[b]) => [b] *)
    function
       egcd :: "(width => ((bvec * bvec * bvec * bvec) => bvec))" where
      "egcd e (a, ra, b, rb)
         = (if b = (0 :w (e + 1))
            then ra
            else (egcd e)
                 (if (even e) b
                  then (a, ra, cshiftr b (1 :w (max e 1)), half rb)
                  else if a < b
                       then (a, ra, cshiftr (b - a) (1 :w (max e 1)),
                             half ((field_sub c) (p, rb, ra)))
                       else (b, rb, cshiftr (a - b) (1 :w (max e 1)),
                             half ((field_sub c) (p, ra, rb)))))"
    by pat_completeness auto
    declare egcd.psimps[simp del]
  end
```

The initial comment names the three nested functions that have been translated to the locale (`even`, `half` and `egcd`), displays the location of their definitions in the Cryptol source file, and also for each one lists a reason why it needed to be lifted to a locale. In this case the nested definition `egcd` is recursive and requires a locale to house a top-level definition, and since both `half` and `even` are reachable from `egcd` they were added to the locale. The name of the locale is derived from the name of the top-level function plus the names of the nested functions, and the c and p locale variables correspond to the extra variables that are in scope inside the nesting. Inside the locale the definitions have exactly the same form as translated Cryptol programs, except that the definitions are allowed to reference the c and p locale variables.

The second Isabelle declaration that the `field_div` Cryptol function translates to is an *interpretation* of the locale:

```
interpretation field_div_egcd_even_half_locale:
field_div_egcd_even_half for c p .
```

Before this locale interpretation declaration the locale was a merely a template; after this declaration the template has been instantiated and the definitions inside are available to the theory.

The third and final Isabelle declaration that is the result of translating the `field_div` Cryptol function is the definition of a `field_div` function:

```
(* ["field.cry", line 57, col 1] *)
(* field_div : {a} (fin a,a >= 1) => ([a],[a],[a]) -> [a] *)
fun
   field_div :: "(width => ((bvec * bvec * bvec) => bvec))" where
  "field_div c (p, x, y)
     = (((field_div_egcd_even_half.egcd c p) (c + (-1)))
        (p, 0 :w c, y, x))"
declare field_div.simps[simp del]
```

It follows the logic of its source Cryptol function by calling the nested `egcd` function that has been made available by the locale.

3.3 Formalizing the Program Correctness Property

Recall from Chapter 1 that the two main use cases for the Cryptol to Isabelle translation are: (i) proving equivalence between two Cryptol programs; and (ii) proving properties of one Cryptol program. In the case of `field_div`, its correctness is expressed by the one property that multiplying its result by the non-zero second argument (the denominator) should always yield the first argument (the numerator). There is no reference implementation that we can prove is equivalent to `field_div` as an alternative to showing that it satisfies its correctness

property—indeed, it is not even easy to prove that the division x/y exists for every numerator x and non-zero denominator y when calculating modulo an odd prime.

Therefore we simply create an Isabelle goal that converts the arguments and result to integers and checks the correctness property holds in this domain:

```
theorem field_div_correct:
 "[| prime (to_int p);
     odd (to_int p);
     to_int x < to_int p;
     to_int y < to_int p;
     0 < to_int y |] ==>
  (to_int (field_div c (p,x,y)) * to_int y) mod (to_int p) =
  to_int x"
```

Note that this goal makes use of integer operations (such as `mod` and `prime`) defined by standard theories of the Isabelle distribution. Using standard definitions whenever possible helps to give us confidence that we have faithfully formalized the correctness properties of translated Cryptol functions.

3.4 A Divide-and-Conquer Proof Strategy

At this point we have completed the translation of the Cryptol program and set the goal corresponding to its correctness property: what remains is to use Isabelle to interactively construct a proof of the goal and turn it into a theorem. This activity is no different in nature from standard formalizations using Isabelle: breaking down the goal into subgoals using tactics; while building up useful lemmas from the supporting theories. The goal is proved when the lemmas match the subgoals.

Note that the goal contains symbols from both the supporting Cryptol theories (e.g., `Bvec`) and standard Isabelle theories (e.g., `Primes`). This is typical for goals corresponding to properties of translated Cryptol programs. One proof strategy that has been effective is to define a new version `field_div'` of the translated Cryptol program `field_div` that uses only symbols from standard Isabelle theories, and thus factor the interactive proof effort into two stages:

1. Prove that converting the result of `field_div` to standard Isabelle theories is the same as converting the arguments of `field_div'`, i.e.,

   $$\texttt{to\_int}(\texttt{field\_div}(p, x, y)) = \texttt{field\_div'}(\texttt{to\_int}(p), \texttt{to\_int}(x), \texttt{to\_int}(y))$$

   This half of the verification makes use of the supporting Cryptol theories (such as `Bvec`), but is simplified by the new version `field_div'` having exactly the same structure as the original `field_div`.

2. Prove that the new version `field_div'` of the Cryptol program satisfies the correctness property. This half of the verification is simplified by the goal containing no symbols from the supporting Cryptol theories, only the standard Isabelle theories.

In the case of `field_div`, the second half of the verification proved to be most challenging, especially reasoning about a standard version of the nested `egcd` function. A major step was proving the following induction scheme that covers all coprime inputs a, b to `egcd`:

```
lemma egcd_induct:
  fixes phi :: "nat => nat => nat => nat => nat => nat => bool"
  fixes p :: nat
  assumes "prime p"
  assumes "odd p"
  assumes "phi p 1 ra b rb ra"
  assumes "phi p a ra 1 rb rb"
  assumes "[| coprime (2 * a) b;
              phi p a ra b rb g |] ==>
           phi p (2 * a) ((2 * ra) mod p) b rb g"
  assumes "[| coprime a (2 * b);
              phi p a ra b rb g |] ==>
           phi p a ra (2 * b) ((2 * rb) mod p) g"
  assumes "[| coprime a b;
              phi p a ra b rb g |] ==>
           phi p (b + a) ((rb + ra) mod p) b rb g"
  assumes "[| coprime a b;
              phi p a ra b rb g |] ==>
           phi p a ra (a + b) ((ra + rb) mod p) g"
  shows "coprime a b ==> phi p a ra b rb (egcd p a ra b rb)"
```

The step cases of the induction scheme match the tail-recursive structure of the `egcd` function, and because the initial value of the a input is a prime the coprimality of the a and b arguments is preserved like a loop invariant. From this we can derive other properties relating the result to the arguments, including the correctness property of `field_div`.

This concludes the finite field division example, which illustrates the complexities that emerge when using Isabelle to verify realistically-sized Cryptol programs, and techniques for understanding and handling them.

Chapter 4

Scalar Multiplication of Elliptic Curve Points

CHAPTER GOALS:

- See a realistic case study of an algorithmic optimization that is outside the scope of automatic proof tools.
- Understand verification techniques for factoring Isabelle equivalence proofs into compositions of smaller proofs.

In the previous chapter we verified that a Cryptol implementation of an efficient finite field division algorithm satisfied a mathematical correctness property. In this chapter we will consider the other main use case of the Cryptol to Isabelle translator: proving the equivalence of two Cryptol programs.

4.1 Reference Implementation

This example we consider comes from an elliptic curve cryptography library, and computes the multiplication of an elliptic curve point by a scalar. To explain the algorithm, we first need some context. For a fixed elliptic curve over a finite field, it is possible to define Cryptol functions that add, subtract and double points on the curve:

$$\begin{aligned} \texttt{point\_add}(P, Q) &= P + Q \\ \texttt{point\_sub}(P, Q) &= P - Q \\ \texttt{point\_double}(P) &= 2P \end{aligned}$$

These all take approximately the same time to execute, so we will generically refer to the cost of any of them as a *point addition*.

Scalar multiplication takes a natural number n and an elliptic point P, and returns

$$nP = P + \cdots + P$$

(i.e., the sum of n copies of P). The efficiency of a scalar multiplication implementation is dominated by the number of point additions it requires.

We create a reference version of scalar multiplication by defining a Cryptol function that simply calls the `point_add` function n times to directly sum n copies of the point P:

```
reference_point_mult : ([384],Point) -> Point;
reference_point_mult (n,p) =
  if n == 0 then point_zero
  else point_add(reference_point_mult(n - 1, p), p);
```

The scalar is represented as a bitvector of length 384, because in this context of our library we know that the scalar multiplication function will always be called with a scalar smaller than 2^{384}. The `point_zero` value represents the distinguished identity point of the elliptic curve.

4.2 Optimized Implementation

Although the reference implementation above has the virtue of being clearly correct, its performance makes it completely unusable in practice. An optimized algorithm that is frequently used is the *repeated squaring* method:

```
repeated_squaring_point_mult : ([384],Point) -> Point;
repeated_squaring_point_mult (n,p) =
  if n == 0 then point_zero
  else if n @ 0 then point_add(repeated_squaring_point_mult(n - 1, p), p)
  else point_double(repeated_squaring_point_mult(n >> 1, p));
```

On average this reduces the number of point additions that are required to $\frac{3}{2}\log_2 n$. However, this is not the best that we can do. Routine 2.2.10 in *Mathematical routines for the NIST prime elliptic curves* [5] computes scalar multiplication using a *signed bit* encoding of the input scalar, and on average requires $\frac{4}{3}\log_2 n$ point additions.[1] Here is a Cryptol implementation of signed bit scalar multiplication:

[1] On average this breaks down as $\log_2 n$ calls to `point_double`, plus $\frac{1}{6}\log_2 n$ each to `point_add` and `point_sub`.

```
signed_bit_point_mult : ([384],Point) -> Point;
signed_bit_point_mult (n,p) = res @ (384+1)
  where {
    k : [384 + 2];
    k = n # [False False];

    h : [384 + 2];
    h = k + (k + k);

    res : [inf]Point;
    res = [point_zero]
          #  [| if hi & ~ki then
                  point_add(point_double(r), p)
                else if ~hi  & ki then
                  point_sub(point_double(r), p)
                else
                  p384_ref_double(r)
             || hi <- reverse(h) # zero
             || ki <- reverse(k) # zero
             || r  <- res |];
    };
```

4.3 Proving Equivalence in the Isabelle Theorem Prover

The reference version and signed bit version of scalar multiplication implement different algorithms, thus putting an equivalence proof out of the reach of existing automatic proof tools. Thus the verification is a good candidate for interactive proof using Isabelle.

The first step is to apply the translator to generate an Isabelle theory containing both the reference and signed bit Cryptol implementations. There are no surprises here: all nested Cryptol functions are converted to Isabelle locales, just as we saw in Chapter 3. Since the reference and signed bit encodings have the same Isabelle type, the goal claiming their equivalence is simply:

```
theorem signed_bit_reference:
 "signed_bit_point_mult = reference_point_mult"
```

How can we interactively prove this equivalence in Isabelle? The answer is always the same: break the goal down into subgoals using tactics; build up useful lemmas from the supporting theories; repeat until the lemmas match the subgoals. The Isabelle goals that come from verifying Cryptol programs can be successfully tackled using standard iteractive proof

techniques. Notwithstanding, in the remainder of this section we present a brief overview of our verification structure for interactively proving the above goal, in the hope that the reader will see common patterns that can be applied to other verifications.

Abstract lower-level functions We first note that both the reference and signed bit implementations call the point operation functions, and it occurs to us to ask what properties of these lower-level functions are required for the implementations to be equivalent. A sufficient property is that the point operation functions plus the identity point form a group. We can use this to factor the verification into: (i) proving the point operations satisfy the group laws; and (ii) proving the equivalence of the reference and signed bit implementations over an arbitrary group. Note that although (ii) is a logically stronger subgoal than the original goal, it is paradoxically easier to prove because the complexity of the elliptic curve has been abstracted away.

Factor out the support Cryptol theories This step is described in detail in Section 3.4, and here we apply it to both the reference and signed bit implementations, to replace the Cryptol bitvector type in the scalar input with the Isabelle natural number type. This again improves our ability to interactively prove the tool, because many standard Isabelle tactics are primed with knowledge of how to simplify and deduce facts about built-in types such as natural numbers.

Create an explicit signed bit encoding type After the above abstraction steps, we are left with the essential core of the verification. Observe that the signed bit implementation combines two phases: (i) construct a signed bit encoding of the input scalar n by comparing the bits of n and $3n$; and (ii) call the addition, subtraction and double point operations based on the sequence of signed bits. We make this explicit by defining an explicit type of signed bits in Isabelle, together with a function `fromNat` for converting natural numbers to sequences of signed bits and a function `exp` for interpreting sequences of signed bits as group operations. This factors the verification one last time into: (i) prove that signed bit implementation is equivalent to the function composition `exp` $\circ$ `fromNat`; and (ii) prove that the function composition is equivalent to the reference implementation.

Bibliography

[1] Levent Erkök. Equivalence and Safety Checking in Cryptol.

[2] Galois, Inc. *Cryptol Programming Guide*, October 2008. Available for download at `http://corp.galois.com/storage/files/downloads/Cryptol_ProgrammingGuide.pdf`.

[3] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2003.

[4] J. R. Lewis and B. Martin. Cryptol: High assurance, retargetable crypto development and validation. In *Military Communications Conference 2003*, volume 2, pages 820–825. IEEE, October 2003.

[5] National Security Agency. Mathematical routines for the NIST prime elliptic curves. Available for download at `http://www.nsa.gov/ia/_files/nist-routines.pdf`, April 2010.

[6] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, October 2011. Available for download at `http://isabelle.in.tum.de/dist/Isabelle2011-1/doc/tutorial.pdf`.