Pragmatic Equivalence and Safety Checking in Cryptol

Levent Erkök John Matthews

Galois, Inc.
421 SW 6th Ave. Suite 300
Portland, OR 97204
{levent.erkok,matthews}@galois.com

Abstract

Cryptol is programming a language designed for specifying and programming cryptographic algorithms. In order to meet high-assurance requirements, Cryptol comes with a suite of formal-methods based tools allowing users to perform various program verification tasks. In the fully automated mode, Cryptol uses modern off-the-shelf SAT and SMT solvers to perform verification in a push-button manner. In the manual mode, Cryptol produces Isabelle/HOL specifications that can be interactively verified using the Isabelle theorem prover. In this paper, we provide an overview of Cryptol's verification toolset, describing our experiences with building a practical programming environment with dedicated support for formal verification.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms Languages, Reliability, Verification

Keywords Cryptography, Formal methods, Equivalence checking, SAT/SM solving, Size polymorphism, Theorem proving

1. Introduction

Cryptol is a domain specific language tailored for cryptographic algorithms (`www.cryptol.net`). Explicit support for program verification is an indispensable part of the Cryptol toolset, due to the inherent high-assurance requirements of the application domain. To this end, Cryptol comes with a suite of formal-methods based tools, allowing users to perform various program verification tasks.

In this paper, we provide an overview of the Cryptol language and its verification environment. The challenges in this domain are multifaceted: from the engineering concerns of providing an easy-to-use system for non-experts, to open research problems in program verification. We will explore Cryptol's verification framework, technologies employed, and the research challenges remaining.

2. A taste of Cryptol

Cryptol is a pure functional language, built on top of a Hindley-Milner style polymorphic type system (Hindley 1997), extended with size-polymorphism and arithmetic type predicates (Lewis and Martin 2003). The size-polymorphic type system has been designed to capture constraints that naturally arise in cryptographic specifications. To illustrate, consider the following text from the AES definition (NIST 2001, Section 3.1):

> The **input** and **output** for the AES algorithm each consist of **sequences of 128 bits**. ... The **Cipher Key** for the AES algorithm is a **sequence of 128, 192 or 256 bits**. Other input, output and Cipher Key lengths are not permitted by this standard.

This description is captured precisely in Cryptol by the following type:

```
encrypt: {k} (k >= 2, 4 >= k)
              => ([128], [64*k]) -> [128]
```

Anything to the left of `=>` are quantified type-variables and predicates on them. In this case, the function `encrypt` is size-polymorphic, relying on the size-variable `k`. The predicates constrain what values the quantified size-variables can take on: Here, `k` is restricted to be between 2 and 4. To the right of `=>`, we see the actual type. The type of `encrypt` is a function, from `([128], [64*k])` to `[128]`. The input to `encrypt` is a pair of two words, the first of which is precisely 128-bits wide; this is the plain-text. The second argument is a `64*k`-bit wide word (i.e., 128, 192, or 256 bits, depending on `k`), which corresponds to the key. The output of the function, the ciphertext, is another 128-bit word.

Note how this type precisely corresponds to the English description in the standard. For instance, it is statically ensured that the key-size will be one of 128, 192, or 256: the type system would not let a definition of `encrypt` be accepted that did not satisfy this requirement. (Similarly, all call-sites will be checked to satisfy this requirement as well.) Cryptol's expressive power in stating precise size relationships is one of its key advantages. In other languages, such constraints would either only appear in comments, or at best get dynamically checked at run-time, instead of being enforced statically at compile-time as in Cryptol.

Predicates and type-expressions in Cryptol can specify arbitrary arithmetic constraints on the quantified variables, including non-linear operations. For instance, here is a padding function that adds enough 0's to a bit-stream to make its length a multiple of 32:

```
pad32 : {a} (fin a, (32*((a+31)/32)) >= a)
          => [a] -> [32*((a+31)/32)];
pad32 bs = bs # zero;
```

Notice how the arithmetic expressions describe the type, and the operation of `pad32`, very precisely.[1]

PLPV'09, January 20, 2009, Savannah, Georgia, USA.

[1] The `fin` predicate states that the input must be a positive integer that is provably finite. (Cryptol can deal with infinite streams as well, a crucial

Having such a precise type pays off nicely in the definition of `pad32`: It simply appends (the `#` operator) the polymorphic constant `zero` (which is the shape-polymorphic sequence of 0-bits) to its input. The actual number of bits to append is encoded in the polymorphic type, and hence need not be explicitly computed by the user. (It is an interesting exercise to compare this definition to an implementation in a language like C.) By lifting such size constraints to types, Cryptol allows both for very precise specifications that are enforced by the compiler, and much simpler implementations that are easier to write, understand, and maintain.

While this flexibility is crucial for expressive power, it comes with a price: Full type-checking for Cryptol would require a system where arbitrary arithmetic predicates over integers with non-linear operators can be proven equivalent, which is an undecidable problem (Tarski et al. 1953). As a practical consequence, Cryptol can sometimes fail to discharge predicates that express mathematical tautologies. For instance, the reader can verify that the constraint

```
32*((a+31)/32) >= a
```

that appears in the type of `pad32` is in fact unnecessary, since it holds for all possible values `a` can take. (Size variables range over non-negative integers.) While Cryptol can discharge many such constraints using an internal arithmetic solver, it does not always succeed in doing so, causing tautological predicates to show up in function signatures and corresponding call-sites. Although these predicates are harmless from a theoretical point of view, they might be confusing for unsuspecting users.

3. Verification of Cryptol programs

Cryptol's program verification framework has been designed to address equivalence and safety checking problems.

The equivalence-checking problem asks whether two functions f and g agree on all inputs. In case f and g are not equivalent, we would like to find a particular value x such that $f\ x \neq g\ x$.

The safety-checking problem is about run-time exceptions. Given a function f, we would like to know if f's execution can perform operations such as division-by-zero or index-out-of-bounds. If so, we would like to know what particular input values will result in an exceptional behavior. (We will discuss safety-checking in detail in Section 6.)

Users interact with Cryptol's verification system via `theorem` declarations. Here is an example Cryptol theorem, named `C`, stating that `dec` and `enc` are cryptographic inverses of each other:[2]

```
theorem C: {key pt}.
      dec(key, enc(key, pt)) === pt;
```

Theorems in Cryptol are simply bit-valued functions, taking their quantified variables as inputs, returning `True` or `False` as their result. We call this the theorem-function correspondence. This unified view provides a consistent and familiar environment for end-users, avoiding the need for an extra language to express properties. As a nice consequence, proving a Cryptol theorem simply amounts to showing that it is equivalent to the constant function that always returns `True`. This is the main mechanism that connects program verification to equivalence checking in Cryptol.

As the reader might suspect, there are no known methods for automatically proving arbitrary Cryptol theorems (or performing safety checks). Furthermore, even when an automated proof might be possible, time/memory requirements might make the approach infeasible. Therefore, Cryptol provides two main verification routes:

- *Fully-automated:* For a restricted (but relatively large) subset of Cryptol, both equivalence and safety checking problems can be decided. For this subset, Cryptol provides fully automated tools to prove theorems with no further user involvement.
- *Semi-automated:* When automated solvers are not suitable, Cryptol provides a tool to translate Cryptol theorems to Isabelle/HOL automatically, where proofs can be manually constructed by the user.

It is a fair question to ask why Cryptol's formal-methods tools only focus on safety and functional-equivalence checking, instead of adapting a more general logic for expressing and proving properties. The reasons are quite practical:

- To do equivalence checking, users need not learn any new language in which to express properties. The term language of Cryptol is sufficient to express functional correctness properties concisely.
- Equivalence checking works well with an incremental development model; successive versions of an algorithm can be proven equivalent to a known "reference" specification, following the stepwise-refinement approach.
- Due to the theorem-function correspondence, all Cryptol theorems are executable. We take advantage of this correspondence by providing a "quick-check" facility, where theorems are checked automatically over randomly generated input data to aid in high-assurance development (Claessen and Hughes 2000).
- Counterexamples provided by equivalence checkers are extremely important. Having a concrete "bug" to look at makes all the difference in providing a practical verification environment for end-users.

Also, on the social side, we have found that the equivalence checking approach lowers the barriers for the adoption of formal methods. Since the users are not required to write program properties in an unfamiliar logic, equivalence checking is easier to introduce into daily development activities. The automated quick-check facility provides instant feedback, empowering users to try alternate ideas with a certain degree of confidence with no extra cost. Additionally, theorem declarations mixed with source-code serve as valuable documentation that is guaranteed not to become obsolete as the code around them evolves.

Equivalence checking applies not only to user programs, but also to compiler generated code as well. Cryptol's FPGA compiler performs extensive and often very complicated transformations to turn Cryptol programs into hardware primitives available on target FPGA platforms. The formal verification framework of Cryptol also allows equivalence checking between Cryptol and netlist representations that are generated by various parts of the compiler. Therefore, any potential bugs in the compiler itself are also caught by the same formal verification framework. This is a crucial aspect of the system: proving the Cryptol compiler correct would be a prohibitively expensive task, if not impossible. Instead, Cryptol provides a *verifying* compiler, generating code together with a formal-proof that the output is functionally equivalent to the input program.

4. Automated equivalence checking

The main idea behind automated equivalence checking is to translate Cryptol into a much simpler *symbolic bit-vector* (SBV) lan-

feature for encoding bit-stream ciphers.) The arithmetic division (`/`) operator returns the quotient, ignoring the remainder. To compute the next size from `a` that is a multiple of 32, we simply add 31 and divide by 32; determining the number of 32-bit chunks we need to have in the result. The final bit-size is simply 32 times the number of chunks.

[2] The notation "`{key pt}.`" reads "for all values of `key` and `pt`."

guage, for which there are existing decision procedures. This translation is done via symbolic execution. To illustrate, let us consider a very simple cipher, which simply adds/subtracts a key:[3]

```
enc, dec: {a b} ([b], [a][b]) -> [a][b];
enc (key, pts) = [| pt + key || pt <- pts |];
dec (key, cts) = [| ct - key || ct <- cts |];
```

We would like to prove theorem `C` from Section 3 for this encryption/decryption pair, proving that they are inverses of each other. In this case, theorem `C` has the following polymorphic type:

```
C : {a b} ([a],[b][a]) -> Bit
```

As we shall see in Section 7.1, we can only prove theorems at monomorphic instances. So, let us fix `a` to be `8` and `b` to be `2`, and prove `C` at the following monomorphic type:

```
C : ([8],[2][8]) -> Bit
```

That is, the key and plain-text are each 8-bits wide; and we are interested in encrypting/decrypting 2 blocks at a time. The symbolic interpreter executes `C` over symbolic variables, obtaining the following SBV program:

```
INPUT s0:[8]
INPUT s1:[8]
INPUT s2:[8]
s3:[8] = (s1:[8] + s0:[8])
s4:[8] = (s3:[8] - s0:[8])
s5:[1] = (s1:[8] = s4:[8])
s6:[8] = (s2:[8] + s0:[8])
s7:[8] = (s6:[8] - s0:[8])
s8:[1] = (s2:[8] = s7:[8])
s9:[1] = (s5:[1] & s8:[1])
OUTPUT s9:[1]
```

The input `key` is represented by the 8-bit symbolic quantity `s0`. The input vector `pts` is represented by two variables, `s1` and `s2`, one for each 8-bit wide plain-text value. It is easy to see that `s3` and `s6` represent the encrypted text, while `s4` and `s7` represent the results of decryption. The variables `s5` and `s8` check that the final results match the inputs. Hence, proving the theorem amounts to showing that the value of `s9` is always `1` regardless of the values of `s0`-`s2`.

While the details of the SBV format are beyond the scope of this paper, the following points are worth noting:

- The program is completely monomorphised.
- Cryptol types are serialized into of fixed-size bit-vectors. The target language only consists of bit-vector data and arithmetic/logical operations on them, all expressed in a three-address form. In particular, there are no nested expressions.
- All function calls are completely translated away via unrolling. There are no jumps remaining, the output is a straight-line sequence of bit-vector operations. (Termination of the unrolling process is sometimes a problem. We will discuss this issue in Section 7.4.)
- The resulting SBV program is in static-single-assignment form: Each variable s_i is assigned exactly once.

Compiling Cryptol into SBV is only the first (and the easier) part of the verification task. The more challenging task is to show that the final output of an SBV program always equals `1` regardless of its inputs, as we discuss next.

[3] The type `[a][b]` denotes a sequence of `a` elements, each of which is of type `[b]`. The type `[b]`, short-hand for `[b]Bit` in Cryptol, indicates a sequence of `b` bits, analogously.

4.1 SAT based equivalence checking

It is a fairly straightforward task to bit-blast an SBV program to obtain a corresponding bit-level representation. The idea is to represent each bit-vector operation by a series of operations on individual bits, much like how they would be implemented in hardware. In this manner, the SBV-decision problem can be reduced to an instance of SAT, the boolean satisfiability question, using only the bit-and ($\wedge$) and negation ($\neg$) operations.

Once this translation is done, we simply ask a SAT solver whether there are any assignments to input variables such that the output node will ever be 0. If the answer to this questions is "unsatisfiable," meaning there are no such input values, we can conclude that the original theorem is indeed valid. If, on the other hand, there are values for the inputs making the output node 0, we will have found a counterexample to the statement of the theorem, i.e., a bug.

This approach does indeed yield a decision procedure for the Cryptol equivalence-checking problem by reduction to SAT. Unfortunately, the feasibility of this approach depends on the particular problem at hand, since SAT is NP-complete. On the positive side, research in SAT-solving over the last decade has had several major breakthroughs and many good SAT solvers are now freely available, such as `minisat` (Eén and Sörensson 2003), and `abc` (Mishchenko 2007). Cryptol gets distributed with the `jaig` tool (Nordin 2005), which is built on top of `minisat`, incorporating a number of Cryptol-specific heuristics.

While the problem remains NP-complete, we have found that the SAT-based approach works well for the kinds of structural transformations the Cryptol compiler tends to perform, such as automated pipelining and rewrite-based simplification. For instance, we were able to prove that a 55-stage pipelined implementation of 128-bit AES is equivalent to its (unpipelined) reference implementation in about 30 seconds with the `jaig` tool, and in about 28 minutes with `abc`, both on commodity hardware. (The generated SAT problem in this case had about 1.7-million AND-gates.)

4.2 SMT based equivalence checking

The main limitation of the SAT-based approach is that it ignores the high-level structure of the verification problem. For certain problems, most notably functions involving multiplication, the SAT based approach does not scale well. A recent alternative is to use SMT (satisfiability-modulo-theories) solvers, which come equipped with dedicated solvers for bit-vector problems. The key idea is to delay the bit-blasting process as much as possible, taking advantage of any arithmetic or structural properties that might be present in the problem.

Cryptol has connections to several SMT solvers, including Yices from SRI (Duterte and de Moura 2006) and CVC3 from NYU (Barrett and Tinelli 2008).[4] Similar to the SAT case, Cryptol sends the question if the output can ever be 0 to the SMT solver; for which any potential model generated by the external tool would indicate a bug found.

We have found that SMT-based solvers perform better for handling problems that have a higher-level structure, especially those depending on algebraic equalities. However, many cryptographic algorithms rely on extensive bit-level manipulation and bit-shuffling by their very nature, so we have not seen a dramatic speed-up in using SMT-based solvers for such algorithms. On the other hand, SMT-solvers for the bit-vector theory are still in their infancy. We are encouraged by the results of recent SMT-solver competitions, and we believe that SMT solvers will be an integral part of Cryptol's automatic verification toolbox as they mature.

[4] Cryptol has a generic API to integrate any SMT solver using the SMT-Lib format, so adding new solvers is a straightforward task.

This is especially true as we start tackling problems in elliptic-curve cryptography, which typically rely on high-level algebraic equalities instead of low-level bit-jumbling.

5. Finding satisfying assignments

We have noted in Section 4 that proving a Cryptol theorem amounts to showing it equivalent to the constant function that always returns `True`. There is a nice dual as well: For bit-valued functions, any counter-examples obtained when showing equivalence to the constant function `False` are essentially the satisfying assignments. Cryptol's `:sat` command precisely implements this idea. (In essence, this technique allows expressing a limited form of existentially quantified properties in Cryptol.)

To illustrate, consider how one can formulate the known plaintext attack using a satisfiability query in Cryptol. If we know that algorithm `enc` produces `ct` for input `pt`, we can try to find the `key` using a query of the form:

```
> :sat (\key -> enc (key, pt) === ct)
```

We should emphasize that Cryptol is *not* a crypto-analysis tool: The associated verification tools are not meant to be used for establishing cryptographic strength of algorithms. In fact, the expectation in posing such a query is that the automated solvers will take an infeasible amount of time to answer it! The `:sat` command is most useful in using Cryptol to solve simple constraint satisfaction problems, taking advantage of the power of the underlying SAT/SMT solvers.

Checking satisfiability of a bit-valued function and proving equivalence of two functions are equivalent problems from a theoretical point of view. (Showing `f` equals `g` amounts to showing that the function `\x -> f x != g x` is not satisfiable. In the other direction, proving `f` unsatisfiable is the same as showing that it is equivalent to the function `\x -> False`, as we noted earlier.) However, we have found that supporting both functionalities in Cryptol's formal verification toolbox allows users to express the desired properties in the most natural way.

6. Ensuring safe execution

The other weapon in Cryptol's verification toolbox is the automated verification capability that programs cannot cause run-time exceptions, such as division-by-zero or index-out-of-bounds. If such an exception is possible, Cryptol will display a particular input value that causes the exception. Here is an example to illustrate this facility in action:

```
> :safe (\(x,y) -> x/y:[8])
*** Violation detected:
((\(x,y) -> x/y:[8])) (0x00, 0x00)
         = divide by zero
```

If proper care is taken to avoid such failures, the function will be proven safe:

```
> :safe (\(x,y) -> if y==0 then 1
                           else x/y:[8])
*** Verified safe.
```

The technique for ensuring safe execution relies on generating verification conditions during symbolic execution. As the symbolic interpreter executes, it keeps track of its control path, i.e., all the test expressions of the `if-then-else` expressions it has gone through. Then, for each potentially unsafe operation, it generates a verification condition to ensure that the operand values are safe whenever this particular path is taken.

For instance, in the second example above, Cryptol generates the verification condition `y` $\neq$ `0` for the else-branch, i.e., in the control path where `y == 0` is false, which easily gets discharged by the prover. In the first example, the same verification condition is generated in the empty control path, which cannot be discharged and hence reported to user as a safety violation. Other operations are similar. Index-out-of-bounds exceptions, for instance, causes the symbolic interpreter to generate the verification condition that the value of the index is less than the length of the sequence being indexed in the current control path.

Automated safety-checking addresses an inherent source of software bugs. Unsurprisingly, manually performing such checks becomes simply infeasible as programs get larger. In fact, we were able to identify several safety issues in programs that were manually translated from their original C implementations to Cryptol. Further inspection of the code revealed that the hand-translations were indeed correct; it was the original C programs that had the corresponding safety problems, yet they survived the test suites that were explicitly designed to validate them.

The following exceptions are caught by Cryptol's safety checker: (1) Division and modulus by 0, (2) index-out-of-bounds, (3) calling `lg2` (the base-2 logarithm function) on 0, (4) uses of Cryptol's `error` function and `undefined` value, and (5) failure cases for Cryptol's `ASSERT` expression.

The last two items are especially important. Cryptol's `error` function and the `undefined` value causes a user-generated run-time exception if their value is ever needed. They are typically used by programmers to indicate that an unexpected case has happened. `ASSERT` expressions are similar, they are mainly used for writing invariants. The safety checker ensures either that these cases will never happen, or it will provide a concrete counterexample where the unexpected case does indeed happen, indicating a bug.

The only two other run-time exceptions that are not caught by the safety-checker are the cases where the program runs out of memory or stack. The former can happen in the presence of large data and/or space leaks. The latter can happen if there are non-terminating execution paths. However, if the program is susceptible to these errors, the symbolic execution itself will fail as well. That is, the `:safe` command will fail to complete, alerting the user.

7. Restrictions of the automated system

Having given an overview of Cryptol's automated verification framework, let us now turn to the question of what subset of Cryptol is subject to this analysis. The push-button framework applies only to theorems that are:

- Monomorphic (Section 7.1),
- Finite (Section 7.2),
- First-order (Section 7.3),
- Symbolically terminating (Section 7.4).

The first three restrictions can be solely detected by looking at the type of the theorem. (The verification framework will alert the user if they are violated.) The last restriction is undecidable in general. In this case the verification process itself might fail to terminate.

7.1 Polymorphism

Polymorphic programming is a key technique for developing reusable components. In the case of Cryptol, size-polymorphic definitions are typically used to encode algorithms that work uniformly over multiple key/block sizes, as we have seen in Section 2. Naturally, properties about such functions themselves tend to be polymorphic as well.

Unfortunately, proving polymorphic theorems is beyond the capabilities of the automated verification system. The backend de-

cision procedures that perform the verification task only work on monomorphic programs. Therefore, only particular monomorphic instances can be automatically verified, as we have explained in Section 4.

The question arises, then, whether a proof at a particular monomorphic type is sufficient to conclude polymorphic validity. The answer is, unfortunately, no. While this intuition does hold for many practical cases, we can surely write theorems that are only valid at certain monomorphic types. Here is a simple example:

```
theorem P: {x}. x*x <= x+x+x;
```

The inferred type of `P` is: `{a} (fin a) => [a] -> Bit`.[5] The validity of this theorem depends on what bit-size it is instantiated at. Here are two particular cases:

- `a = 2`: In this case the arithmetic is done modulo $2^2 = 4$. The possible values for `x` are `0`, `1`, `2`, and `3`. The user can easily verify that the statement is indeed a theorem for all these input values.
- `a = 3`: The arithmetic is now done modulo $2^3 = 8$. The variable `x` ranges from `0` to `7`. It is easy to see that when `x = 6`, the theorem states `4 <= 2`, which is false.

This observation leads to the first restriction on the automated prover: Only theorems at a monomorphic type can be proven. Validity of polymorphic theorems cannot be automatically established.

As the reader might suspect, not all polymorphic theorems are size-sensitive in practice. It is an open research problem to devise an algorithm to determine if a given polymorphic Cryptol theorem is size-independent in this sense.

7.2 Dealing with infinite streams

Functions that accept and return infinite streams are not supported by the automated verification system. There is simply no way to represent an arbitrary infinite stream symbolically. (Recall that SBV programs have only bitvector data of arbitrary-but-fixed sizes.)

The most interesting case is functions that return an infinite stream, typically used to represent sequential output in Cryptol. To illustrate, consider the following theorem:[6]

```
theorem I: {x}. [y (y+1) ..] === [z (z+1) ..]
  where { y = x+x; z = (2:[8])*x };
```

Theorem `I` states that the infinite sequences

```
[(x+x) (x+x+1) (x+x+2) ..]
```

and

```
[(2*x) (2*x+1) (2*x+2) ..]
```

are equivalent. Since Cryptol's equivalence checker cannot symbolically execute an infinite sequence producing function, we will have to restrict ourselves to only finite segments of the output:

```
theorem I': {x}.
   take (100, [y (y+1) ..])
      === take (100, [z (z+1) ..])
  where { y = x+x; z = (2:[8])*x };
```

In general, proving equivalence over an infinite stream of output would require induction, which is beyond the capabilities of Cryptol's push-button equivalence checking system.

7.2.1 Counter based attacks

While proving equivalence for a segment of length k might provide some evidence of correct behavior, it does not provide a proof of correctness. In the particular domain of cryptography, such proofs will fail to identify malicious code based on counters. The idea is simple: A malicious implementation of an encryption algorithm could deliberately "behave" correctly for k cycles, and start to leak the secret key afterwards. To illustrate, let `enc` be an encryption algorithm. Here is a definition for `enc'`, which starts leaking the key after one thousand cycles of correct execution:

```
enc' (key, pts) = take(1000, enc (key, pts))
                # [key key ..];
```

A theorem stating `enc` and `enc'` behave the same for any finite prefix will fail to identify the malicious code, unless the prefix size happens to be larger than one thousand.

7.2.2 Refactoring

In certain cases, it might be possible to refactor a function producing an infinite stream into a form that is easier to verify automatically. Let us reconsider theorem `I` from Section 7.2. It is easy to see that we can use a seed-value/next element encoding to implement its components:

```
gen1, gen2 : [8] -> ([8], [8] -> [8]);
gen1 s = (s+s, \v -> v+1);
gen2 s = (2*s, \v -> v+1);
```

The first element of the output tuple corresponds to the first element of the output, while the second element is the function capturing how successive elements are related. We can easily use the equivalence checker to prove equivalence of these two components separately.

How do we use this state machine representation in a real program? Such a machine can be turned into an infinite stream producing function as follows:

```
m2s : {a b} (a -> (b,b -> b)) -> a -> [inf]b;
m2s f = \x -> res
 where { (seed, next) = f x;
         res = [seed]
             # [| next r || r <- res |]
       };
```

Note that `m2s` itself will not be part of the verification path; we will only have to work with `gen1` and `gen2` in the proof.

While refactoring may first seem to be a tedious task, it is usually an exercise that is well worth the effort. Notice that only the top-level infinite-stream producing function we are equivalence checking has to be refactored this way. (Using infinite streams elsewhere in the program is actually an encouraged Cryptol idiom that is fully supported by the equivalence checker.) Furthermore, such functions typically follow a well-defined encryption mode such as cipher-block-chaining or counter modes (Dworkin 2001). Therefore, the rewrite needed is generally rather rudimentary.

7.3 Higher-order functions

A higher-order function takes or returns another function as an argument (such as the `m2s` function from Section 7.2.2). Equivalence checking in the presence of higher-order functions is an undecidable problem in general. Unsurprisingly, our symbolic execution based equivalence checking framework does not support such functions, as there is no data-type to map them to in the SBV language.

[5] The `fin` (finite) predicate comes from the use of arithmetic operations and comparison. All arithmetic in Cryptol is modular, with modulus 2^n where n is the size of the arguments.

[6] Cryptol's enumeration syntax `[a b ..]` represents the infinite sequence starting with `a`, successively incrementing the previous element by $b - a$ at each position, with wrap-around using modular arithmetic.

Techniques such as firstification/defunctionalization might prove helpful in dealing with such problems, should one need to work with higher-order theorems (Reynolds 1972).

7.4 Symbolic termination

The final restriction for automated verification in Cryptol is the requirement of symbolic termination: All recursive calls in a given function must terminate with respect to the initial set of symbolic inputs.

To illustrate, consider the following function `sum`, which computes the sum of numbers up to `n`, modulo 8:

```
sum : [8] -> [8];
sum n = if n == 0 then 0 else n + sum (n-1);
```

The function `sum` is *not* symbolically terminating: The base case of its recursive definition relies on a non-static check. That is, in order to stop the recursive call, we have to compare the symbolic input variable `n` against `0`, which cannot be done statically. It is important to note that the function `sum` is terminating for all (concrete) inputs. Symbolic termination is a stronger property than termination.

Here is how one can rewrite `sum` so that it is symbolically terminating:

```
sum' : [8] -> [8];
sum' n = s (n, 0)
 where s (n, mx) =
           if (n == 0) | (mx == 255)
           then 0
           else n + s (n-1, mx+1);
```

The `mx` argument keeps track of how deep we can recurse. We start by calling `s` with the constant value of `0` for `mx`, which is incremented at each recursive call. At the 256th unfolding of `s`, the value of `mx` will be 255. Therefore the expression `(n == 0) | (mx == 255)` will be reduced to `True` by the symbolic simulator, even though `n` is symbolic. Therefore, the unfolding will stop, terminating the translation process. (Contrast this to the original definition of `sum` above, where the test of the `if` expression always remains symbolic, causing the symbolic simulator to run indefinitely.) This alternative definition of `sum` can now be used for automated equivalence checking since the termination will be controlled by a static argument.

Unfortunately, the symbolic termination requirement is not something Cryptol can check ahead of time and warn the user against, as opposed to all the other restrictions. On the other hand, symbolic termination is a rare issue in Cryptol, since recursive functions are typically discouraged and rarely needed in practice. (Also see Section 10.1 for a technique to remove this restriction completely.)

8. Verification via theorem proving

Cryptol's automated verification system works in a complete push-button manner, providing a seamless integration between daily programming and formal verification. Unfortunately, the fully automated system is not the panacea. Some important Cryptol theorems can not be proved in a reasonable amount of time (i.e. several days) by any of the SAT or SMT solvers currently available. These problems typically rely on high-level algebraic equivalences, and involve sub-problems that are notoriously hard for SAT/SMT solvers, such as multiplication.

Large-word modular multiplication underlies many public-key crypto algorithms such as RSA and ECC (Elliptic Curve Cryptography). A modular multiplication algorithm computes $u \times v \mod m$ where u, v, and m are all unsigned bitvectors of hundreds or thousands of bits wide. However, using the SAT and SMT solvers currently available to us we cannot verify even simple implementations of ordinary bitvector multiplication ($u \times v$) at these word sizes, much less modular multiplication. In fact, equivalence checking even simple multiplication properties such as $(a \times b) \times c = a \times (b \times c)$ is known to take time exponential to the word size of the variables involved, at least when using the current SAT based technologies (Kroening and Strichman 2008, Section 6.3.1).

To deal with these difficult theorems we have recently added a translator from our SBV format to the Isabelle proof assistant (Nipkow et al. 2002). Isabelle has a wide array of existing rewrite and proof search methods, and also allows users to certify their own custom proof strategies. The use of a proof assistant such as Isabelle allows one to integrate human insight into the proof process when fully automated tools do not provide a feasible alternative.

8.1 Example Isabelle proof

To illustrate the use of Isabelle in equivalence checking, let us reconsider theorem `C` from Section 4. Below is the Isabelle translation of this theorem, automatically generated by Cryptol:

```
theory Example imports SBV
begin
consts v0 :: 8 word
consts v1 :: 8 word
consts v2 :: 8 word

definition v3 :: 8 word where v3 = v0 + v1
definition v4 :: 8 word where v4 = v3 - v0
definition v5 :: 1 word where
  v5 = bool2bv (v1 = v4)
definition v6 :: 8 word where v6 = v0 + v2
definition v7 :: 8 word where v7 = v6 - v0
definition v8 :: 1 word where
  v8 = bool2bv (v2 = v7)
definition v9 :: 1 word where
  v9 = (v5 AND v8)

lemmas sbv_defs =
  v3_def v4_def v5_def v6_def
  v7_def v8_def v9_def

lemma match: v9 = (1 :: 1 word)
oops (* User's proof script goes here *)

end
```

In the Isabelle/HOL translation the symbolic SBV inputs `s0`, `s1`, and `s2` become uninterpreted constants `v0`, `v1`, and `v2` of the theory, and each SBV definition becomes a corresponding defined constant. Each Isabelle `definition` statement also generates a corresponding theorem asserting that the left- and right-hand sides of the definition are equal. For user convenience during proofs, these theorems are collected up into a named list of lemmas called `sbv_defs`. Finally, we add a postulated lemma `match` stating that the output variable `v9` is always `1`, regardless of the values of the uninterpreted constants `v0`, `v1`, and `v2`. The `oops` proof command on the next line tells Isabelle to abort the proof attempt and not record `match` as a proved theorem.

At this point the user must step in and replace `oops` with an appropriate proof script. This is where the user intervention comes into play. As we shall see shortly, the following simple script is sufficient in this case:

```
apply (unfold sbv_defs)
apply simp
done
```

The command `unfold sbv_defs` expands all definitions referred to in `sbv_defs`, resulting in the proof state

```
goal (1 subgoal):
 1. bool2bv1 (v1 = v0 + v1 - v0) AND
    bool2bv1 (v2 = v0 + v2 - v0) = 1
```

The command `simp` can now solve this subgoal directly, using arithmetic simplification rules and rewrite rules for `bool2bv1` that we have previously installed in Isabelle's database of default rewrites. The new proof state becomes:

```
goal:
No subgoals!
```

Since the proof is now finished, the final command `done` instructs Isabelle to record the original formula as a certified lemma.

8.2 Verifying large-word multiplication

We have also used Isabelle to prove theorems that could not be handled by our existing automated tools. For example, we have verified a simple implementation of a large-word multiplier, establishing that it corresponds to Isabelle's definition of multiplication. In this case we had to prove auxiliary lemmas that allowed us to rewrite all bitvector expressions in terms of modular arithmetic on unbounded integers. We were then able to carry out the verification using Isabelle's existing database of modular arithmetic lemmas as well as a library of new modular arithmetic rewrites we proved correct. (This library is shipped with Cryptol to aid end-users construct similar proofs.)

Because the multiplier implementation had been completely unrolled by the SBV translator, the intermediate Isabelle terms grew quite large. To prevent these terms growing even larger and swamping the simplifier, we had to develop a custom rewriting strategy that carefully staged the order in which rewrite rules were applied. Isabelle was then able to verify a 256-bit implementation of the multiplication problem in 25 minutes, and a 384-bit implementation in about 2.5 hours, which are both beyond the capabilities of currently available SAT/SMT solvers.

8.3 Isabelle limitations

Although these timings are far better than we could get from current push-button tools, they do not scale arbitrarily. In particular, Isabelle's simplifier represents all terms as trees. However, the original SBV expressions actually represent a directed-acyclic-graph (DAG), and usually have a lot of subterm sharing. Expanding this DAG representation into a tree can easily cause an exponential blowup in the term sizes. There have been Cryptol theorems we could not verify using Isabelle's current simplifier because of this blowup.

It is possible to certify a custom simplifier in Isabelle that would preserve SBV's DAG representation. However we have not done this yet. Instead, we are planning to add rewrite strategies directly into DPT, an open source SMT solver (Goel et al. 2008). Here we hope to gain the best of both worlds, combining the efficient DAG term representation, case splitting, and backtracking strategies of an SMT solver with the higher level rewrite strategies available in proof assistants such as Isabelle.

To ensure that our combined solver is still sound, we also plan to leverage a proof logging facility that is currently being added to DPT. We would replay DPT proof logs in Isabelle, being careful to maintain the proof's DAG term structure. Isabelle can already replay very large SAT proofs using similar techniques (Weber and Amjad 2007).

9. Related work

Equivalence checking has a long history in the hardware design community, and is one of the flagship applications of formal-methods in industry (Huang and Cheng 1998). Equivalence checking software against software, and software against hardware are also very active research topics (Feng 2007). We review some of the most relevant work below, mostly from a programming language/theorem proving point of view.

9.1 Safety checking stream-transformers

Cryptol is an example of a *stream transformer* language, where time-dependent behaviors—wires—are explicitly represented as streams (infinite sequences). Stateful programs are then represented as pure functions that take and deliver streams.

Lustre is a commercial stream transformer language that targets safety-critical reactive programs (Halbwachs et al. 1991). Lustre is the core language of the SCADE toolset, provided by Esterel Technologies, Inc. SCADE comes with a *sequential safety checker*, which can check that a property holds of every element of a Lustre infinite stream. In contrast, our safety checker can only check these kinds of properties for a fixed, finite prefix. Full sequential safety checking has not been a priority for us, since crypto-algorithms are designed to terminate after a fixed number of iterations, as otherwise they would be vulnerable to timing attacks.

Recently Hagen and Tinelli have also implemented a sequential safety checker for Lustre named *Kind*, using Yices as its underlying SMT-solver (Hagen and Tinelli 2008).

As far as we know, there are no equivalence checkers for Lustre, either bounded or sequential.

9.2 Interactive theorem proving

Many cryptographic algorithms have been specified in the formal logics of interactive theorem provers (Toma and Borrione 2005; Duan et al. 2005). This approach has the benefit of leveraging a very powerful verification platform for proving both safety and equivalence properties, at the cost of requiring expert human guidance to carry out the proofs.

9.3 Shallow embedding into higher-order-logics

Various subsets of Cryptol have been embedded in higher-order logics before, for the purposes of modeling and theorem proving (Li and Slind 2005; Hurd 2007). In each case, termination was a big issue since the target logics (close variants of HOL) only admit total functions. The other main difficulty was in finding a faithful embedding with respect to Cryptol's size-types. In Li and Slind's approach, lazy lists are used for representing all sequences. This allows for a flexible means for manipulating them easily, but loses size type information. In Hurd's approach, functions from naturals represent infinite sequences, while functions from singleton-numeric types are used to represent finite ones. Although Hurd's approach is more precise, it does not handle size-polymorphic Cryptol functions that accept both finite and infinite sequences, due to the fundamental difference in how these two types are represented.

9.4 Verifying compilers

Theorem provers have also been used as *verifying compilers* to automatically prove that a compiler has generated correct code for a given crypto specification (Pike et al. 2006; Slind et al. 2007). This is in contrast to a *verified compiler* which has a once-and-for-all proof that the compiler itself is correct. We use our equivalence checker as a limited form of a verifying compiler for Cryptol's VHDL synthesizer: at various stages of the compiler we can output a circuit representation of the compiler's intermediate representation language. We symbolically simulate each circuit for a fixed

number of clock cycles, and then check that the result is equivalent to what the original Cryptol specification says the output should be at that particular clock cycle.

10. Future work

There are a number of research opportunities/challenges regarding equivalence and safety checking in Cryptol. In this section, we will briefly describe some of the most promising possibilities.

10.1 Precise termination analysis

Recall from Section 7.4 that the automated verification system requires all recursive functions to be symbolically terminating. In fact, this restriction is not fundamental and can be lifted completely. The problem arises since the symbolic simulator takes a conservative approach when dealing with if-expressions: Unless the test expression statically evaluates to `True` or `False`, it simply expands both branches. An alternative approach would be to check whether it is possible to have a `True` or `False` value computed instead. If one of these values is not possible, then we can simply eliminate that branch. For instance, in the `sum` example from Section 7.4, we can statically determine that the if-expression will necessarily evaluate to `True` at the 255th unfolding of the recursive call, allowing us to stop the infinite unfolding at that point. Since all data domains are finite in Cryptol, we can always decide this question using a SAT/SMT solver during the translation process itself.

While the implementation of precise termination analysis is not particularly hard, it would be best to use an incremental solver for performing the termination queries. (An incremental solver is one where satisfiability checks can be done incrementally, allowing one to add further constraints in between these checks.) Unfortunately, neither the SMT-Lib language we are using for communicating to SMT-solvers, nor the AIG format we use for communicating to SAT-based solvers allow incremental queries. With a non-incremental solver, we would be running similar queries over and over, which would dominate analysis time, making the overall approach impractical.

We do have plans for a much tighter integration of Cryptol with incremental SMT solvers, and in particular with DPT (Goel et al. 2008). Once this incremental bridge is available, we plan to implement precise termination analysis to increase the subset of Cryptol the automated verifier can handle, essentially lifting the symbolic-termination requirement.[7]

10.2 Size polymorphism

Recall from Section 7.1 that validity at a monomorphic instance does not necessarily imply validity for polymorphic theorems. The question we would like to investigate is whether we can come up with a list of conditions that would be sufficient to validate a polymorphic theorem from a monomorphic instance proof. It would still be very useful even if we can only come up with a conservative answer. (That is, answering "unknown" when we cannot decide.) Designing such an algorithm for Cryptol remains an open research problem.

10.3 Verification with infinite sequences

The other research area for Cryptol equivalence checking is performing proofs for functions producing infinite sequences, typically used in modeling sequential output over time. As we have discussed in Section 7.2, an inductive proof would be needed for addressing this problem in general. Our current push-button equivalence checking framework lacks this capability. However, if we can identify common themes (such as encoding crypto-modes), we might be able to simplify the problem significantly.

10.4 Identifying malicious code

Unsurprisingly, not all properties of interest can be cast as functional equivalence problems. This is especially true in the domain of cryptography, where non-functional security requirements are inherent. For instance, if we are handed an alleged FPGA implementation of AES, we would like to ensure that it not only implements AES correctly, but also that it does not contain any "extra-circuitry" to leak the key. In general, we would like to show that an end-user cannot gain any information from an implementation that cannot be obtained from a reference specification. It is an open research question to design algorithms that can effectively detect such side-channel attacks for Cryptol.

10.5 More general translation to Isabelle/HOL

Recall from Section 8 that our current translation of Cryptol into Isabelle/HOL goes through the SBV format, which implies definitions are monomorphised and completely unrolled. While this unrolling is necessary when SAT/SMT solvers are targeted, it is not always the best choice when targeting Isabelle, which has a much richer programming environment. To this end, we have recently started building a translator from Cryptol to Isabelle/HOL that maintains the polymorphic typing and the recursive structure of programs, motivated by earlier work along these lines (Li and Slind 2005; Hurd 2007). We are planning to use this embedding to carry out general correctness proofs manually, especially when extra assurance is needed. However, as we have pointed out earlier, the differences between Cryptol's and HOL's type systems complicate the Isabelle/HOL translation significantly, requiring more research in this area to ensure a sound and effective embedding.

11. The trusted code base

Cryptol's formal verification system inherently relies on the correctness of Cryptol's compiler front-end components (i.e., the parser, the type system, etc.), the symbolic simulator, and the translators to SAT/SMT solvers and to Isabelle/HOL. Note that Cryptol's internal compiler passes, optimizations, and code generators (i.e., the typical compiler back-end components) are *not* in the trusted code base.

While Cryptol's trusted code base is only a fraction of the entire Cryptol tool suite, it is nevertheless a large chunk of Haskell code. Needless to say, we heavily test this trusted code base using an extensive regression test suite.

12. Conclusions

In this paper, we have described the verification framework of Cryptol, a key ingredient of Cryptol's high-assurance development environment. The system has already proved itself useful in practice, especially in establishing the equivalence of reference and pipelined implementations of crypto-algorithms, essential in generating efficient FPGA representations. Since such transformations are done both manually by programmers and automatically by the compiler, it is absolutely essential to ensure that correctness is preserved through many structural transformations applied to obtain a time/space efficient realization.

Developing a practical programming language with formal verification in mind is an inherently difficult task. The needs for "ease of programming," and "ease of verification" are often at odds. Supporting more language constructs to simplify programming comes with an obligation to support those constructs precisely in the proof

[7] Note that termination will always remain a requirement. If a recursive function does *not* terminate on all of its inputs, then the symbolic simulator will simply loop forever. Handling such functions in general would amount to solving the halting problem.

environment as well. (In the case of Cryptol, this is further complicated by the desire to support compilation down to FPGA based targets, where "run-time" resources are extremely limited.) Nonetheless, the rather general programming model of Cryptol based on size-types and pure functional programming, along with being explicit about what is supported for automated verification paid off nicely in practice, balancing the needs of both sides.

Acknowledgments

Many people have worked on Cryptol and its formal verification toolset over the years, including Jeff Lewis, Thomas Nordin, and Phil Weaver. In particular, Thomas Nordin has developed Cryptol's SAT based equivalence checker `jaig`.

We are also thankful to Magnus Carlsson, Lee Pike, John Launchbury, Joe Hurd, Sally Browning, and the anonymous PLPV reviewers for their feedback and comments on this paper.

References

Clark Barrett and Cesare Tinelli. CVC3 web site. `cs.nyu.edu/acsys/cvc3`, 2008.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000.

J. Duan, J. Hurd, G. Li, S. Owens, K. Slind, and J. Zhang. Functional correctness proofs of encryption algorithms. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, December 2005.

Bruno Duterte and Leonardo de Moura. The YICES SMT Solver. Available at: `yices.csl.sri.com/tool-paper.pdf`, 2006.

Morris Dworkin. Recommendation for block cipher modes of operation: Methods and techniques, December 2001. URL `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`. NIST Special Publication 800-38A.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. ISBN 3-540-20851-8.

Xiushan Feng. *Formal equivalence checking of software specifications vs. hardware implementations*. PhD thesis, Vancouver, BC, Canada, Canada, 2007.

Amit Goel, Jim Grundy, and Sava Krstić. DPT web site. `http://sourceforge.net/projects/dpt`, 2008.

George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In A. Cimatti and R. Jones, editors, *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD'08), Portland, Oregon*, Lecture Notes in Computer Science. Springer, 2008. URL `ftp://ftp.cs.uiowa.edu/pub/tinelli/papers/HagTin-FMCAD-08.pdf`.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

J. Roger Hindley. *Basic Simple Type Theory*, volume 42. Cambridge University Press, Cambridge, UK, 1997.

Shi-Yu Huang and Kwant-Ting Cheng. *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

Joe Hurd. Embedding Cryptol in higher order logic. Available from the author's website, March 2007. URL `http://www.gilith.com/research/papers`.

Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

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.

Guodong Li and Konrad Slind. An embedding of Cryptol in HOL-4. Unpublished draft, 2005.

Alan Mishchenko. ABC: System for sequential synthesis and verification. Release 70930, Available at: `http://www.eecs.berkeley.edu/~alanmi/abc`, 2007.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

NIST. Announcing the AES, November 2001. URL `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`. FIPS Publication 197.

Thomas Nordin. The `jaig` Equivalence Checker. Available from Objective Security Company, 2005.

Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 1–10, New York, NY, USA, 2006. ACM. ISBN 0-9788493-0-2. doi: http://doi.acm.org/10.1145/1217975.1217977.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM. doi: http://doi.acm.org/10.1145/800194.805852.

Konrad Slind, Scott Owens, Juliano Iyoda, and Mike Gordon. Proof producing synthesis of arithmetic and cryptographic hardware. *Formal Aspects of Computing*, 19(3), August 2007.

A. Tarski, A. Mostowski, and R. M. Robinson. *Undecidable Theories*. North-Holland Publishing Company, 1953.

Diana Toma and Dominique Borrione. Formal verification of a SHA-1 circuit core using ACL2. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2005. ISBN 3-540-28372-2.

Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, July 2007. URL `http://dx.doi.org/10.1016/j.jal.2007.07.003`.