A Verifying Core for a Cryptographic Language Compiler[*]

Lee Pike
leepike@galois.com

Mark Shields[†]
mbs@cartesianclosed.com

John Matthews
matthews@galois.com

Galois Connections
Beaverton, Oregon, USA

ABSTRACT

A *verifying compiler* is one that emits both object code and a proof of correspondence between object and source code.[1] We report the use of *ACL2* in building a verifying compiler for *μCryptol*, a stream-based language for encryption algorithm specification that targets Rockwell Collins' *AAMP7* microprocessor (and is designed to compile efficiently to hardware, too). This paper reports on our success in verifying the "core" transformations of the compiler – those transformations over the sub-language of *μCryptol* that begin after "higher-order" aspects of the language are compiled away, and finish just before hardware or software specific transformations are exercised. The core transformations are responsible for aggressive optimizations. We have written an *ACL2* macro that automatically generates both the correspondence theorems and their proofs. The compiler also supplies measure functions that *ACL2* uses to automatically prove termination of *μCryptol* programs, including programs with mutually-recursive cliques of streams. Our verifying compiler has proved the correctness of its core transformations for multiple algorithms, including TEA, RC6, and AES. Finally, we describe an *ACL2* book of primitive operations for the general specification and verification of encryption algorithms.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods, reliability*; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms

security verification reliability cryptography

Keywords

ACL2, verifying compiler, certifying compiler, optimizing compiler, high-assurance, certification, cryptography

[*]The *ACL2* books associated with this paper can be retrieved at ⟨`http://www.galois.com/files/core_verifier/`⟩.

[†]Present Address: Microsoft, Redmond, Washington, USA.

[1]Our use of the term "verifying compiler" differs from Tony Hoare's use of it in describing his "Grand Challenge" [10]. However, the fundamental goal of increased software assurance via proof is shared by a verifying compiler (in our sense) and the Grand Challenge. Henceforth in this paper, "verifying compiler" should be understood in our sense only.

1. INTRODUCTION

A *high-assurance compiler* is a compiler associated with some strong body of evidence for its correctness. High-assurance compilers are sought in the domain of cryptography, in which they are *security-critical* to their users. An incorrect design could prevent communication between two parties or allow a malicious intruder to intercept encrypted communication, leading to loss of property or even life. Furthermore, some customers of cryptographic compilers consider their cryptographic algorithms to be classified, in effect placing a wall between the end-users and the developers of the language and compiler. The customers may even consider the developers to be untrusted (e.g., if the customers and developers are citizens of different nations), and therefore require assurance that a compiler does not contain malicious code (e.g., a back door).

The evidence provided to demonstrate a compiler is high-assurance can vary in both kind and strength. Some examples of evidence for compiler correctness include widespread and longterm use without incident (such as with the *gcc* compiler for *C*), substantial testing, certification by some authority (e.g., Common Criteria certification [3] or FAA software certification [6]), or as we consider, a mathematical proof of correctness.

A proof provides a strong form of assurance that complements and may even subsume traditional testing and manual-inspection methods. Widespread and long-term use as well as testing can miss bugs when compiling unconventional but well-formed language constructs or when exotic optimizations are enabled. Certification oftentimes attempts to ensure a high-quality *process* for software development is followed, relying on the tacit assumption that a good process more likely yields a good product. Mathematical proof, on the other hand, ensures that requirements and assumptions for correct behavior are made explicit, and a proof is about (a model of) the artifact itself rather than the design process.

There are two approaches for providing assurance of compiler correctness via mathematical proof. The first approach is to build a *verified compiler* in which the transformations of the compiler are proved to be correct for all (well-defined) programs. A more modest approach is to build a *verifying compiler*.[1] A verifying compiler is not itself verified, but each compilation it performs is. That is, given a program, a verifying compiler produces two outputs: (1) a compiled program, and (2) a proof of correspondence between the object code and the source code. Some of the difficulties in producing a verifying compiler are orthogonal to producing a verified compiler. A verified compiler requires a monolithic proof of correctness. The proof is deep and difficult. On the other hand, the proofs associated with a verifying compiler are much shallower. A proof generated by a verifying compiler need only demonstrate correspondence for a concrete program.

The essential difficulty with building a verifying compiler is that the verification must be automated. The compiler must generate both the correspondence theorem statement and its proof with little or no human guidance. The problem of automation, for a production-quality compiler and real-world programs, is the central one addressed in this paper.

For the proofs associated with either a verified or verifying compiler, a formal verification tool, such as a mechanical theorem-prover, is needed. For a verified compiler, a proof of correctness will be large and tedious. A mechanical theorem-prover can assist in proof management as well as ensuring no corner-case is glossed over. For a verifying compiler, a formal verification tool is essential; paper and pencil proofs are not automated.

In this paper, we show how the *ACL2* mechanical theorem-prover [12] can be used to build the verification infrastructure for the core transformations of a compiler for *μCryptol* (pronounced "micro cryptol"), a derivative of *Cryptol* [25]. Like *Cryptol*, *μCryptol* is designed for the specification of symmetric-key cryptographic algorithms, but *μCryptol* and its compiler, `mcc`, are particularly designed for the efficient compilation to both hardware and embedded microprocessors without dynamic memory management (the 'μ' should not be taken to imply that *μCryptol* is a "toy" language; rather, the language should be considered to be a full-featured derivative of *Cryptol* particularly designed for compilation to embedded software platforms as well as hardware).

We have developed a verification infrastructure for the core transformations of the *μCryptol* compiler. The full infrastructure is depicted in Figure 1 and discussed in detail in Section 2. Briefly, the portion of the infrastructure described herein begins at the point at which the language constructs of *μCryptol* can be *shallowly embedded* into *ACL2*; most notably, we begin after pattern matching has completed and true infinite streams have been transformed into *indexed form* in which each (possibly nested) stream is represented as a top-level function from indexes to values, as described in Section 3. The core transformations are responsible for aggressive target-independent optimizations, and they end just before architecture-specific (i.e. software vs. hardware) transformations are exercised. At this point the program is in *canonical form* (also described in Section 3). Our *ACL2* macro automatically generates a termination proof for both the indexed form and the canonical form of the *μCryptol* program, and then proves that the two forms are input-output equivalent.

We have demonstrated our approach by verifying the output generated for a variety of simple *μCryptol* programs (e.g., the Fibonacci and factorial functions) as well as the more substantial TEA [29], RC6 [23], and AES [5] encryption algorithms, all of which are included in the associated *ACL2* books.[2]

In summary, this paper and its associated files make the following contributions:

- A methodology for building a verifying compiler for *μCryptol* using *ACL2* including a methodology for generating automated equivalence proofs between (stylized) recursive programs and their iterative forms.

- A framework and translator for the shallow embedding of *μCryptol* into *ACL2*.

- A method for automatically generating *ACL2* measure functions and automated termination proofs in *ACL2* of (potentially) mutually-recursive stream definitions from the *μCryptol* well-definedness algorithm [25].

- An *ACL2* book of executable primitive operations useful for specifying symmetric key encryption algorithms (including modular arithmetic, arithmetic in Galois Fields, bitvector operations, and vector operations).

The remainder of the paper is organized as follows. Section 2 overviews the infrastructure of the verifying portion of the compiler and makes more precise what our correspondence theorems mean. The *μCryptol* language and the `mcc` compiler are described in Section 3; we give a feel for the language by describing the specification of the factorial function. We particularly focus on establishing well-definedness of cliques of mutually-recursive streams and the core transformations of the compiler. In Section 4, we begin by describing the embedding of *μCryptol* into *Common Lisp*, focusing particularly on embedding types and on the *ACL2* book of *μCryptol* primitive operations. We then describe automated termination proofs in *ACL2* for mutually-recursive cliques of streams and finally the automated generation of both theorem statements and proofs in *ACL2*. Section 5 describes related work, and concluding remarks are given in Section 6.

Familiarity with *ACL2* and *Common Lisp* will help the reader but is not required to understand the majority of our presentation.

2. VERIFYING COMPILER OVERVIEW

2.1 Proof Infrastructure

Figure 1 shows the overall proof architecture of the verifying *μCryptol* compiler. The architecture is split into three blocks, respectively labeled *front-end*, *core*, and *back-end*. These blocks correspond to the architecture of both language compilation and verification. To avoid confusion, we call the portion of the verifying compiler that compiles the language the *compiler* and the portion that carries out the correspondence proofs the *verifier*. The second author, who independently designed *μCryptol* and implemented `mcc`, describes the compiler in detail elsewhere [25], while we focus

[2]The associated books currently depend on the Rockwell Collins' `super-ihs` book, which is slated for public release.

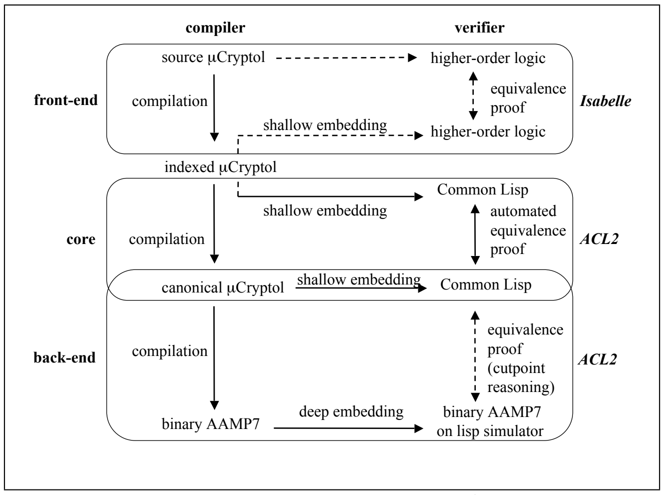


Figure 1: Verifying Compiler Proof Architecture

on the verifier in this paper. In the figure, dashed lines represent in-progress portions of the verifying compiler, whereas solid lines represent finished work. For completeness, we briefly describe the verifier in full below, and make a few comments about in-progress work in Section 6.

To begin, a *shallow embedding* from language L_1 to language L_2 is a specification of the semantics of some well-formed formula (WFF) of L_1 in the syntax of L_2; a shallow embedding is just a translation between languages. A shallow embedding contrasts with a *deep embedding* in which the abstract syntax of L_1 is represented in L_2 (usually by an algebraic datatype of L_2), and a *meaning function* is constructed that maps instances of the datatype to WFF of L_2.

The following overviews the three portions of the verifier.

- **Front-End**: Source *μCryptol* is a higher-order language, which precludes us from shallowly embedding it into *ACL2*'s first- order logic. Thus the front-end compiles *μCryptol* programs to a first-order *indexed form*, which can then be shallowly embedded into *ACL2*. To verify these front-end transformations, we are starting on another translator that shallowly embeds both original *μCryptol* source and indexed form programs into higher order logic. We will then use the Isabelle [20] higher order logic theorem-prover to automatically prove that any original *μCryptol* program is observationally equivalent to its indexed form.

- **Core**: We have implemented a translator that shallowly embeds indexed and canonical *μCryptol* programs into *Common Lisp*. Both indexed and canonical *μCryptol* are sublanguages of *μCryptol*, so one translator is sufficient for this task.

- **Back-End**: The verifying compiler for the back-end of `mcc` yields a correspondence between canonical *μCryptol* programs and their compiled *AAMP7* machine code. The `mcc` compiler has a translator that outputs a *Common Lisp* list of numbers representing *AAMP7* opcodes. The list is given its intended semantics by Rockwell Collin's state-machine model of the *AAMP7*, written in *ACL2* [8]. Thus, the translation of the machine code is a deep embedding in which the *AAMP7* model acts as the meaning function. The model can be used in two ways. First, the model has been designed to be highly efficient for simulation [13, pp. 89-106], so *ACL2* can simulate the *AAMP7* with the *μCryptol* program loaded. Second, a correspondence can be proved between the result of executing the machine code on the *AAMP7* and the canonical *μCryptol* from which it is compiled.

```
fac : B^32 -> B^8;
fac i = facs @@ i
  where {
    rec
      index : B^8^inf;
      index = [0] ## [ x + 1 | x <- index];
    and
      facs : B^8^inf;
      facs = [1] ## [ x * y | x <- facs
                            | y <- drops{1} index];
  };
```

Figure 2: Factorial Function in *μCryptol*

2.2 Correctness Properties Proved

Here we make precise what the correctness theorems we produce mean. Leroy lists a number of potential correctness properties in compiler verification [14]. Of these, ours is Leroy's condition that for source program S and compiled code C, "if S has well-defined semantics (does not go wrong), then S and C are observationally equivalent." In our case, we take the source program to mean the *μCryptol* program in indexed form, and we take the compiled code to be the *μCryptol* program in canonical form. As mentioned, indexed and canonical forms are sublanguages of *μCryptol*, so they share the same semantics.

Furthermore, since we automatically prove termination of both indexed and canonical forms, our proof of correspondence is a *total correctness* proof.

3. THE LANGUAGE AND COMPILER

3.1 The Language

We introduce the language *μCryptol* by specifying the factorial function, as shown in Figure 2. The *μCryptol* language is similar to (but not a subset of) *Cryptol* [16], which in turn is heavily influenced by Haskell [22]. Due to space considerations, our example does not illustrate the following additional features of *μCryptol*: `if-then-else` conditionals, enumerations, type abbreviations, the application of its type-completion algorithm, constant declarations, tuple constructions, and irrefutable pattern-matching.

The statement `fac : B^32 -> B^8;` declares `fac` to be a function accepting a 32-bit word and returning an 8-bit word. Applying the function to an input that is not expressible in 32 bits results in an error. Nothing is special about word sizes in *μCryptol*; we could have just as easily used 3 or 3177-bit words. Furthermore, a single *μCryptol* program may manipulate words or vectors (words are simply vectors of bits in *μCryptol*) of mixed widths.

Next comes a function definition. *μCryptol* is a first-order language and does not have λ-expressions; hence, all functions must have named definitions, and all function arguments must appear in the definition header. The body of `fac` is motivated by our desire to compile efficiently to both hardware and software. A long tradition in hardware modeling is to represent latches by the infinite sequence of values they latch [24, 11]. We call infinite sequences *streams*, to distinguish them from vectors, whose width must always be finite. By using streams, *μCryptol* avoids needing imperative features, which (1) simplifies both its formal specification and compilation, (2) allows the language to be easily compiled to both hardware and microprocessors, and (3) makes the behavior of programs more predictable by their authors.

The body of `fac` is the value `facs @@ i` such that `facs` is a stream and the `@@` operator takes a 0-based index into a stream. The stream `facs` is in the scope of two nested definitions introduced by a `where` clause (definitions in *μCryptol* may be arbitrarily nested). Just as for the top-level, each definition in a `where` clause is in scope of all following definitions in the same clause, as well as the term to the left of the `where`. In this case, the `where` clause contains a single clique of recursive stream definitions. The keywords `rec` and `and` indicate the beginning and continuation of a clique, respectively.

We begin by describing the stream `index`. Its type signature is `index : B^8^inf`, indicating `index` is a stream of 8-bit words. We use `^inf` as the type constructor for streams, which is suggestive of streams being vectors but with infinite width (`inf` is not, however, a type). The body of `index` defines it to be the singleton vector containing an 8-bit zero (`[0]`) followed by the stream `[x + 1 | x <- index]` (`##` primitive appends a vector to a stream). This term is a *stream comprehension*, and mimics the set comprehensions familiar from mathematics. In particular, the comprehension states "construct a stream which, for each successive element of `index` (call it `x`), has the element `x + 1`." Thus, `index` is the stream 0, 1, 2, 3, 4, . . . , 255, 0, 1, ...

There is a recursive reference to `index` in the stream comprehension, so care must be taken to ensure the definition is *well-defined*. In this case, the second element of `index` depends on only the first element, which is known to be zero. Similarly, the third element depends only on the second, which we just calculated, and so on. A novelty of *μCryptol* is its use of the type system to decide well-definedness of streams. We shall return to well-definedness in Section 3.2.

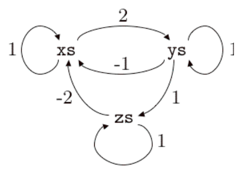
Continuing with the example, we consider the stream `facs`. Also like `index`, its successive elements are defined by a stream comprehension, but this time with two *arms*. A multi-arm stream comprehension steps through each arm in parallel. The first arm draws successive elements from the stream `facs` while the second arm draws elements from the stream `drops{1} index`. This term is the stream `index`, but with the first element removed. Thus, the stream `facs` is 1, 1, 2, 6, 24, 120, 208, 176, . . . and `facs @@ 3` = 6, for example (recall that 0! = 1, by convention).

3.2 Well-defined streams

The `mcc` compiler is able to use the type system of the language to ensure well-definedness of mutually-recursive cliques of streams. Furthermore, the measure function generated by `mcc` to ensure well-definedness is used by *ACL2* to prove termination, as described in Section 4.2. The definition depends on the notion of an integer *stream delay*. We say the "delay from stream x to occurrence of stream y is d" to mean, for sufficiently large index $k \in \mathbf{N}$, that the k'th element of stream x depends on the value of the $(k - d)$'th element of stream y at that occurrence. The "minimum delay from x to y" is the least delay over all occurrences of y, and dually for "maximum delay". The definition is as follows:

```
rec (xs : B^8^inf) =
  [3] ## [ x+y | x<-xs | y<-[0{8}]##ys ];
and (ys : B^8^inf) =
  [5] ## [ x+y+z | x<-drops{2} xs
                 | y<-ys | z<-zs ];
and (zs : B^8^inf) =
  [7] ## [ x+z | x<-drops{3} xs | z<-zs ];
```

Figure 3: An example clique of streams to illustrate well-definedness.



$$f(k, x) = 3 \times (k + \mathit{offset}(x)) + \mathit{order}(x)$$
$$\text{where } \mathit{offset} = \texttt{xs} \mapsto 0, \texttt{ys} \mapsto 1, \texttt{zs} \mapsto 2$$
$$\mathit{order} = \texttt{xs} \mapsto 0, \texttt{ys} \mapsto 2, \texttt{zs} \mapsto 1$$

Figure 4: Minimum Delay Graph for the Streams in Figure 3 and the Corresponding Measure

Definition 1. Let S be the set of stream names defined by a mutually-recursive clique of stream definitions. Then we say the clique is *well defined* if there exists a *measure function*

$$f \in (\mathbf{N} \times S) \to \mathbf{N}$$

such that for each occurrence of a stream y in the body of the definition of stream x with delay d, we have

$$\forall k \in \mathbf{N}. k \geq d \Rightarrow f(k - d, y) < f(k, x)$$

For example, consider the clique of definitions in Figure 3. The minimum delays for these definitions is presented as a directed weighted graph in Figure 4. These definitions are indeed well-defined and a measure function for the streams is presented in the same figure.

An algorithm has been developed to construct a measure function from any minimum delay graph (to ensure the *μCryptol* type system can determine well-definedness, certain constructs are not allowed in recursive cliques of streams) [25]. Thus, for *μCryptol*, the problem of deciding well-definedness is reduced to finding the delay graph G.

3.3 Core Transformations

Here we describe the core transformations. The main point of this and the following section is to demonstrate that automatically generated correspondence proofs are feasible even in the presence of aggressive compiler optimizations. Indeed, the `mcc` core transformations are powerful enough to reduce some exponential-time indexed form programs to linear-time canonical form programs.

Front-End Transformations

1. Introduce safety checks
2. Simplify vector comprehensions
3. Eliminate patterns
4. Eliminate streams
   - 4.1. Convert to indexed form

Indexed Form Generated

Core Transformations

- 4.2. Push stream applications
- 4.3. Collapse arms
- 4.4. Align arms
- 4.5. Takes/segments to indexes
- 4.6. Convert to iterator form

5. Eliminate simple primitives
6. Eliminate zero-sized values
7. Inline and simplify
8. Introduce temporaries
9. Eliminate nested definitions
10. Share top-level value definitions
11. Box top-level definitions
12. Eliminate shadowing

Canonical Form Generated

Figure 5: Steps to convert type-complete *μCryptol* programs into canonical form.

Figure 5 lists the transformations that take source *μCryptol* programs to canonical form. In the figure, the points at which indexed and canonical forms are generated are respectively noted. Transformations 1 through 4.1 correspond to the front-end transformations of Figure 1, and transformations 4.6 through 12 correspond to the core transformations.

We begin by describing the end goal of the core transformations, canonical *μCryptol*. We then focus on the transformations that eliminate streams, transformations 4.1 through 4.6. The nature of the other transformations are largely straightforward or standard from the compilation of functional programming languages.

Canonical Form. Canonical *μCryptol* is a language subset with a direct operational interpretation, and is intended to be as close to a final implementation as possible while still remaining neutral as to hardware *vs.* software implementation (compare it with the STG code of the GHC *Haskell* compiler[21]).

In canonical *μCryptol*, every value arising during execution is either small enough to pass and construct on a value stack (we call such values *unboxed*), or has an explicitly named variable bound to its value in a `where` clause. All function and primitive arguments are ready to be passed without the need for additional storage to evaluate them. All definitions are at the top-level, eliminating the need for closures. Vector comprehensions are of a form implementable by a for-loop. All partial operations (those which may yield an arbitrary result) are wrapped by an explicit `assume` clause, which makes explicit the implied partiality. No patterns remain, and all function arguments are either wildcards (ignored) or simple variables, which name the argument location. A number of primitives, such as `reverse`, are eliminated since they have simple definitions as comprehensions. No values of zero-sized type, such as `()`, `()^3` or `B^0` are ever constructed, passed as arguments, or returned from functions (the absence of side-effects means such values are uniquely determined from their type alone, and so need no run-time representation). No top-level definition is shadowed, relieving the back-end from having to respect scoping.

One final restriction of canonical *μCryptol* is essential for efficiently compiling to hardware and software. We could interpret recursive streams as lazy lists, and give them a direct operational interpretation using updating closures[21] for software, or latched dataflow networks for hardware. However, for software, this would require the implementation to support dynamic memory allocation, while for hardware, it would distribute latches throughout the circuit, rather than grouping them into shift registers. To avoid this problem, canonical *μCryptol* does not support streams, but instead has a notion of *iterators* (iterators are a part of *μCryptol*, and can appear in source programs). Recall from Section 3.2 that every *μCryptol* stream is guaranteed to require only a finite number of earlier elements to calculate the current element, and that we introduced the term "history width" to refer to (an upper bound on) that number. We implement the history buffer "lazily" using modular arithmetic on its index rather than explicitly shuffling along older elements as new ones are calculated. In hardware, the finite history width property can be exploited to represent the history buffer as a shift register with the same depth as the history buffer width.

Step 4.1: Convert to Indexed Form. Stream constructs are replaced with an equivalent term in *indexed form.* For this step, *μCryptol* supports two additional term forms which (1) define streams as functions from indexes to values, and (2) apply such a function to a particular index. For example, the term `[0{8},1] ## ys` (for stream `ys`) appears in indexed form as `\i.{<2->[0{8},1]@(drop i)|true->ys.i -2}`. The `\` indicates abstraction, while the `.` after `ys` indicates application.

Indexed form constructs are highly stylized, and *μCryptol* does not support general λ-abstractions and application. Abstraction is only over values of a special abstract index type `ind{`w`,` d`,` m`,` l`}`, where w is the width of a concrete index into the stream, d is the delay depth of the context of the abstraction, m is the minimum value for the index (in the above example, this value would be 0 for `i` in the body to the right of `<2`, and 2 to the right of `true`), and l is the level for the stream definition (i.e., the number of `where` clauses separating it from the top-level definition). An abstraction is always paired with a case split on the index.

An application must be of the form $t\,.\,tmvar\ \tau$ where t is a stream expression, *tmvar* is a stream index variable, and τ is an integer, indicating the offset to apply to *tmvar* when indexing into t (the parameter m in the lambda abstraction prevents subtracting too large an offset from the index).

Step 4.2: Push Stream Applications. Stream applications are pushed into `if-then-else` and `where` terms, and each stream definition within a clique is η-expanded [1] so as to begin with an explicit index abstraction. (We use a fresh index variable name which is shared by all streams in a clique.)

```
1. \i.{ <2->0
      | <4->(\j.{ <1->1 | <2->2 | true->3 })
              . i -1
      | true -> (\j.{ <5->4 | true->5 })
                  . i -2 }

2. \i.{ <2->0
      | <4->(\j.{ <2->1 | <3->2 | true->3 })
              . i +0 }
      | true -> (\j.{ <7->4 | true->5 })
                  . i +0 }

3. \i.{ <2->0 | <3->2 | <4->3
      | <7->4 | true -> 5 }
```

Figure 6: Example of Arm Collapsing in Step 4.3

Step 4.3: Collapse Arms. Nested stream abstractions are collapsed into a single level. For example, given the abstraction in (1) of Figure 6, we may correct for the application offsets `-1` and `-2` to yield the abstraction in (2), which in turn may be collapsed into the single abstraction in (3). Notice how some abstraction cases (such as that yielding `1`) can be dropped since they will never fire. Thanks to the η-expansion of step 2, each recursive stream definition will at this point be a single abstraction and case split.

Step 4.4: Step Align Arms. The arms are aligned to eliminate strictly positive index offsets. Recall from Section 3.2 the termination measure for a clique of mutually-recursive stream definitions consists of an offset for each stream and an ordering amongst the streams. We now "slide" each stream according to its offset. This will allow the new elements for each stream to be constructed together.

Consider again the example of Figure 3, which at this point of compilation resembles (1) of Figure 7. The offset for `zs` is determined by the type completion algorithm to be 2. We thus wish to slide the stream `zs` forward by 2 elements, inserting two dummy values (which we take to be `0`) at its head. All references to `zs`, both inside and outside the clique of streams, must be updated to take account of this slide. That is, we must add 2 to the offsets for indexes into `zs` from within the definition of `xs` and `ys`, and subtract 2 from the offsets for indexes into `xs` and `ys` from within the definition of `zs`. After doing this and sliding `ys` by its offset 1, the result is as in (2) in Figure 7.

Step 4.5: Takes/Segments to Indexes. Some primitives, such as `takes` and `segments`, are replaced with uses of `@@` if the sub-vector being extracted is wider than the history with of the stream. For example, `takes{3} xs` will be left as is if `xs` has history width of four or greater, but converted to `[xs @@ 0, xs @@ 1, xs @@ 2]` otherwise.

Step 4.6: Convert to Canonical Form. Each clique of stream definitions is rewritten as a single *iterator definition.* A history buffer for each stream is tupled according to the stream ordering in the clique measure. The arms of each stream in the clique are merged together, and the body term for each arm is tupled, also according to the stream ordering. Each reference to a stream within the clique is of the form $x \,.\, i\ - n$ for stream name x, index variable i (which is now

```
1. rec (xs : B^8^inf) = \i .
     { < 1 -> 3
     | < 2 -> (xs . i -1) + 0
     | true -> (xs . i -1) + (ys . i -2) };
   and (ys : B^8^inf) = \i .
     { < 1 -> 5
     | true -> (xs . i +1) + (ys . i -1) +
               (zs . i -1) };
   and (zs : B^8^inf) = \i .
     { < 1 -> 7
     | true -> (xs . i +2) + (zs . i -1) };

2. rec (xs : B^8^inf) = \i .
     { < 1 -> 3
     | < 2 -> (xs . i -1) + 0
     | true -> (xs . i -1) + (ys . i -1) };
   and (ys : B^8^inf) = \i .
     { < 1 -> 0
     | < 2 -> 5
     | true -> (xs . i +0) + (ys . i -1) +
               (zs . i +0) };
   and (zs : B^8^inf) = \i .
     { < 2 -> 0
     | < 3 -> 7
     | true -> (xs . i +0) + (zs . i -1) };
```

Figure 7: Streams of Figure 3 Before (1) and After (2) Step 4.4

shared by all streams in the clique) and natural number n. Each such term is replaced by `project` m j `@ (drop(`i - n`))`, where m is the number of streams in the clique and j the position of stream x in the stream ordering. References to a stream from outside the clique are replaced with a call to the iterator function, a project, and an index. A term such as `takes xs == [ 3, 3, 8, 28, 111 ]` in the scope of these definitions would be translated to

```
[ project 3 0 (iter_xs_ys_zs i)@drop i
| (i : B^32) <- [0, 1 .. 4] ] ==
    [ 3, 3, 8, 28, 111 ]
```

4. THE VERIFIER

4.1 Shallow Embedding

The `mcc` compiler contains a translator that translates the indexed and canonical forms into *Common Lisp*. In the shallow embedding, for all type-correct inputs (according to the *μCryptol* type system), the outputs are equivalent to those of the *μCryptol* program. The translator is a small (excluding libraries, the translator is approximately 1200 lines of code) stand-alone portion of the compiler that is easy to inspect for correctness (we do not prove its correctness). The translator is written in *OCaml*, like the rest of the compiler. The "camlp4" metaprogramming facility of *OCaml* [15] is also employed to embed *Common Lisp* as a quotable sub-language of *OCaml*. We emphasize that the translator performs no semantic transformations. Thus, the translation is rote – as one would desire in a high-assurance compiler – but we do highlight a few of its aspects, including our handling of types, *μCryptol* primitives, and measure functions for mutually-recursive cliques of streams.

Embedding Types. Recall from Section 3 the discussion of the *μCryptol* type system. The *ACL2* logic is untyped, so we embedded the *μCryptol* types into the *ACL2* logic as predicates that constrain the values of the arguments to the embedded functions. For example, the following embeds the type `B^32^2`, the type of vectors of length two containing 32-bit words, and is automatically generated by the translator:

```
(defund |$ind_0_typep| (x)
 (and (true-listp x)
      (natp (nth 0 x))
      (< (nth 0 x) 4294967296)
      (natp (nth 1 x))
      (< (nth 1 x) 4294967296)))
```

Notice that the definition is disabled. In general, we disable type predicates, as their definitions can easily be hundreds of lines of *Common Lisp*. For example, in the shallow embedding of AES, there are type declarations such as `B^8^4^4^11`. The general strategy is to prove rewrite rules for the type predicates of the indexed and canonical outputs, but then leave them disabled in the proofs of correspondence for the other functions.

Primitives. The heart of a domain specific language is a set of primitive operators essential to programming in that domain. The primitives of *μCryptol* are ones that are used to specificy symmetric key cryptographic protocols. We have developed an *ACL2* book implementing these primitives that should prove useful for the specification in *ACL2* of cryptographic protocols, in general. The primitives are documented in the *μCryptol* Reference Manual [26]. The primitives have all been extensively tested against their *μCryptol* implementations. We briefly list the implemented operations below (some of which are simply calls to operations previously defined in other *ACL2* books).

- *Arithmetic in* $\mathbf{Z}_{2^n}$ *(arithmetic modulo* 2^n*)*: addition, negation, subtraction, multiplication, division, remainder after division, greatest common divisor, exponentiation, base-two logarithm, minimum, maximum, and negation.

- *Bitvector operations*: shift left, shift right, rotate left, rotate right, append of arbitrary width bitvectors, extraction of n bitvectors from a bitvector, append of fixed-width bitvectors, split into fixed-width bitvectors, bitvector segment extraction, bitvector transposition, reversal, and parity.

- *Arithmetic in* $\mathbf{GF}_{2^n}$ *(the Galois Field over* 2^n*)*: polynomial addition, multiplication, division, remainder after division, greatest common divisor, irreducibility, and inverse with respect to an irreducible polynomial.

- *Pointwise extension of logical operations to bitvectors*: bitwise conjunction, bitwise disjunction, bitwise exclusive-or, and negation bitwise complementation.

- *Vector operations*: shift left, shift right, rotate left, rotate right, vector append for an arbitrary number of vectors, extraction of n subvectors extraction from a vector, flattening a vector vectors, building a vector of vectors from a vector, taking an arbitrary segment from a vector, vector transposition, and vector reverse.

4.2 Automated Termination Proofs

For a given *μCryptol* program, `mcc` automatically generates a measure function in *ACL2*, and from the measure function, *ACL2* proves termination for arbitrary well-defined cliques of mutually-recursive streams. To admit the shallow embeddings into *ACL2*, we must prove termination for both indexed and canonical forms of the clique. The measure for the canonical form is trivial. Recall from Section 3 that in canonical form, cliques are compiled to a function that takes as arguments a tuple of history buffers, an index, and an index limit, and iterates over the index up to the limit value. The measures generated in canonical form simply demonstrate that in recursive calls, the difference between the limit, *lim*, which is the total number of iterations to be taken, and i, the current iteration, should decrease. Thus, for any clique of streams in canonical form, the measure function f is

$$f(i) = \mathit{lim} + 1 - i$$

The more significant challenge is proving termination for cliques in indexed form. The hard part of determining well-definedness is completed by the type inferencing of `mcc` discussed in Section 3. The generated measures are directly translated into a corresponding *Common Lisp* function. For example, for the clique in Figure 3, the measure output by the `mcc` translator is a straightforward *Common Lisp* implementation of the measure from Figure 4.

The *ACL2* termination proofs guarantee that for a concrete program, its clique of streams are well-defined, and it provides evidence that the `mcc` well-definedness algorithm is correct, in general.

4.3 Theorem Generation

For a verifying compiler, both the statements and the proofs of the correspondence theorems must be generated automatically. We describe our approach for the automated generation of the theorem statements in this section, and defer a description of proof generation to the succeeding section. Because the language of *ACL2* is a programming language, one can generate the theorem statements themselves within *ACL2*. This ability greatly simplified our task. The macro `make-thm` generates the theorem statements (as well as additional required function definitions, hints to the prover, logical theory creation, symbols to be bound, etc.). The `make-thm` macro generates six kinds of correspondence theorems:

- *Function correspondence theorems* of non-recursive definitions.

- *Type correspondence theorems* of type declarations.

- *Vector comprehension correspondence theorems.*

- *Stream-clique correspondence theorems* of recursive cliques of stream comprehensions.

- *Vector-splitting correspondence theorems* of type correspondence for vectors that have been `split` into a vector of subvectors.

- *Inlined* `segments/takes` *correspondence theorems* for inlined `segments` and `takes` operators over streams.

```
(make-thm :name |inv-facs-thm|
          :thm-type invariant
          :ind-name |idx_2_facs_2|
          :itr-name |iter_idx_facs_3|
          :init-hist ((0) (0))
          :hist-widths (0 0)
          :branches (|idx_2| |facs_2|))

(make-thm :name |fac-thm|
          :thm-type fn
          :itr-term (|itr_fac| i)
          :ind-term (|ind_fac| i))
```

Figure 8: Macro Calls to Generate the Correspondence Proofs for Factorial (Figure 2)

Some of these correspondence theorems – such as stream-clique equivalence – require supporting lemmas, which are also automatically built by the macro. The macro takes no arguments but it does take a variety of keys depending on the particular correspondence theorem to be generated. In general, the keys are names or terms from the shallow embeddings. For example, to prove the correspondence between the indexed and canonical forms of the factorial function implementation in Figure 2, the calls to `make-thm` in Figure 8 are sufficient. The first call to the macro generates a stream-clique correspondence theorem. In this case, the macro also generates nine lemmas and two corollaries to the main theorem (the number of each is linear in the number of recursively-defined streams in the clique). The second call generates a function correspondence theorem to demonstrate the correspondence theorem for the top-level function definition, `fac`.

The `mcc` compiler does not yet generate the calls to the `make-thm` macro, but doing so would require relatively modest modifications to the *µCryptol* to *Common Lisp* translator.

4.4 Proof Generation

The automatic generation of proofs in a verifying compiler faces two obstacles. First, we must develop a framework that is general enough to be completely automatic for any potential *µCryptol* program. Second, these are real programs generated from a real compiler, so a substantial portion of our effort involves addressing the "scaling problem" for mechanical theorem-proving. We describe these in turn.

The nature of a verifying compiler coupled with the functionality of *ACL2* make the problem of generality tractable. As mentioned in Section 1, it is much easier to prove propositions about concrete programs than parameterized ones. Thus, rather than proving general rewrite rules, the `make-thm` macro generates an instance of the lemma appropriately instantiated by concrete values (e.g., the width of history buffers) from the program being compiled. Thus, we substantially avoid proving deep parameterized theorems in the course of this work (although we do depend on user-supplied books, such as Rockwell Collins' `super-ihs` book, which contains sophisticated theorems about bitvector arithmetic) [9].

With respect to the second obstacle, although the theorems to be proved are not deep, the size of the terms in the theorem statements (or that the terms rewrite to) quickly overwhelmed *ACL2*. The challenge in this case is not mathematical – such as discovering a sufficiently strong invariant – but in discovering the appropriate theorem statement and set of hints to the prover that allow *ACL2* to prove correspondence for non-trivial programs. Our solutions to the scaling problem are well-known in the *ACL2* community; this should be taken as an industrial case-study demonstrating their efficacy for automating the proofs in a verifying compiler.

```
(defthm factorial-invariant
 (implies
  (and (natp i) (natp lim)
       (true-listp hist) (<= i (+ lim 1))
       (equal (nth (loghead 0 i) (nth 0 hist))
              (ind-facs i 'idx))
       (equal (nth (loghead 1 i) (nth 1 hist))
              (ind-facs i 'facs)))
  (and (equal (nth (loghead 0 lim)
              (itr-facs i lim hist)
              (ind-facs lim 'idx))
       (equal (nth (loghead 1 lim)
              (itr-facs i lim hist)
              (ind-facs lim 'facs)))))
```

Figure 9: Factorial Correspondence Theorem (from Figure 2)

For instance, consider the statement of the equivalence theorem between the indexed and canonical forms for a mutually-recursive clique of streams. It is essentially a conjunction stating that the values in the iterator's history buffer correspond to the values computed by the indexed streams. To illustrate, consider the factorial correspondence theorem (Figure 9) originating from the program shown in Figure 2. In the theorem, `hist` is the tuple of history buffers associated with the iterator form. For the factorial function, `hist` is a pair of history buffers, each of which has length 1. The shallow embedding into *Common Lisp* of the history buffer is the list `((#) (#))` containing the values produced by the iterator function `itr-facs` for the streams `idx` and `facs`, respectively. The hypothesis of the theorem contains two equalities between the values produced by the indexed form of the clique of streams and the values in the history buffers at index `i`, modulo the width of the buffers (the function application `(loghead i j)` returns `j` modulo 2^i; see `books/ihs/` of an *ACL2* installation).

Such a theorem is adequate for history buffers containing a small number of mutually-recursive streams with small delays. However, for large history buffers, it is not feasible to prove a similar theorem in which the hypothesis contains equalities for each location in the history buffer. The problem is not the number of conjuncts; rather, it is the size of the terms and the number and complexity of the rewrites to be applied to them. For example, the theorem analogous to the one presented above that is necessary to define key expansion in AES requires reasoning about a history buffer of length 64, requiring 64 conjunctions in the hypothesis. Furthermore, the indexed and canonical definitions are much more complex than are `ind-facs` and `itr-facs`. Our solution is to encode the conjunction of equalities as a recursive function and then develop the necessary set of rewrites to prove the recursive function is a sufficiently strong hypothesis to imply the conclusion. The `make-thm` macro defines such a recursive function and rewrite rules.

Another approach we take to automate large-scale proofs is to iteratively enable function definitions. In carrying out the proofs, we face a tension: powerful books, such as Rockwell Collins' `super-ihs` book, dramatically simplify reasoning about bitvector operators, over which most of the *μCryptol* primitives are defined. However, many of the rewrites are expensive (i.e., they are tried very often or they trigger a large number of other (potentially expensive) rewrite rules to fire; see the *ACL2* documentation on *accumulated persistence* [12]). For example, early attempts to prove the correspondence theorem for TEA in which function definitions are enabled caused the *ACL2* prover to take an inordinate amount of time or simply run out of memory. On the other hand, disabling all functions (or omitting the book altogether) causes proofs to fail.

Our solution is to disable *μCryptol* primitives and other *ACL2* operators (e.g., `update-nth`), initially, and then conservatively open definitions using a cascade of computed hints. The firing of the computed hints is controlled by prioritizing them (see `books/misc/priorities/` in the *ACL2* installation). Furthermore, the hints are allowed to fire only after the proof has become stable under simplification by the rewriter. This technique not only make proofs feasible that otherwise are not, but it obviates the need to determine whether a particular proof will require some definition to expand; if proof has not terminated after simplification at some point, the disabled functions are automatically expanded until it does.

4.5 Applications

The implementation of the approach is approximately 1500 lines of *Common Lisp*, excluding external books. We have demonstrated the feasibility of the approach by verifying the compiler output for simple programs like factorial and Fibonacci and for the symmetric key encryption and decryption protocols TEA, RC6, and AES. The specifications of these protocols use most of the *μCryptol* language constructs, and AES in particular is non-trivial: the shallow embeddings, termination proofs, and correspondence proofs generate approximately 350 definitions, 200 proofs, and approximately 47,000 lines of *ACL2* prover output. The proofs for AES require approximately 15 minutes to build on typical hardware (a G4 Mac with 1 Gigabyte of RAM).

5. RELATED WORK

Dave provides a recent overview of compiler verification [4]. Our work complements recent work by Leroy et. al. describing a *verified* compiler for a subset of *C* to *PowerPC* assembly code [14, 2]. The formal proofs are carried out using the *Coq* mechanical theorem-prover; indeed, *Coq* is used also for programming the compiler by automatic translation to *Caml* code.

With respect to work using *ACL2* specifically in compiler verification, Goerigk demonstrates in *ACL2* how a formally verified compiler can nevertheless contain a Trojan Horse in its executable [13, pp. 201–215]. Moore reports research in building a fully-verified stack [19, 18].

Closely related to our work is a project implementing a verifying compiler to compile cryptographic algorithms to both the *ARM* microprocessor [7] and FPGAs [28, 27]. The compiler is built inside the *HOL4* mechanical theorem-prover, and compilation is by application of derived inference rules. Thus, the verifier is at the granularity of individual transformations, whereas in our approach, the verifier is applied to a set of transformations (e.g., the core transformations) all at once. Also in contrast with our work, the compiler and verifier are inextricable in their framework. The source language is higher-order logic, and for a subset of this language (e.g., tail-recursive functions over simple datatypes), compilation is automatic. Interactive compilation, or *derivation*, is possible from a larger source language that includes a subclass of linear-recursive programs. Like our effort, this is work in progress; TEA has been compiled, but more significant algorithms, like AES, are current challenges.

6. CONCLUSION

The primary objective of this research is to demonstrate the feasibility of building a real verifying compiler for a domain-specific cryptographic language using *ACL2*.

Although our macro automatically proves correspondence and termination for a range of *μCryptol* programs, we do not guarantee completeness of methodology. This is a pragmatic effort, and language constructs may exist for which the macro does not succeed in proving correspondence, but it is relatively easy to extend the `make-thm` macro with additional parameters, lemmas, etc.

In our work, we did not uncover any bugs in the compiler (as expected). The compiler had been thoroughly tested during development, and bugs in symmetric-key encryption algorithm implementations usually are immediately apparent. Nevertheless, recall from Section 1 that a verifying compiler can also be used to ensure that malicious code (e.g., a "back door") has not be introduced into the compiler. Furthermore, a formal proof of correctness may be required for certification under the Common Criteria [3] for a security-critical compiler.

The *μCryptol* verifier is in-progress, but there are ways to reduce the effort required in the remaining portions of the project. The first two portions of the verifying compiler – the front-end end and middle-end – are target-independent. Thus, for any target of `mcc`, the compiler and verifier for these portions can be left unchanged. For the current `mcc` back-end, verification requires demonstrating a correspondence between *AAMP7* machine code and canonical *μCryptol*. For the target of the *AAMP7*, an *ACL2* model exists [8], and recent advances in machine code reasoning in *ACL2* further reduce the required effort [17]. Rockwell Collins and Galois Connections have used these recently-developed techniques to prove the correctness of a hand-coded machine code factorial program on the *AAMP7*, and Rockwell Collins is currently extending and improving these methods [9] and is beginning the verification of `mcc`-generated machine code. Much of this effort in the machine code verification comes from proving the correspondence between the *μCryptol* primitives and their machine code implementations. One way to reduce this effort is to inspect and test the machine code generated by the compiler for the primitives. Then, under the assumption the primitives are correct, have the verifier prove that the generated machine code implements canonical *μCryptol*.

Acknowledgments

We received advice and comments from the following individuals: David Hardin, Eric Smith, and Tom Johnson, Konrad Slind, our anonymous reviewers, and Matt Kaufmann and the other participants in the *ACL2*-Help email list. This work was performed (under subcontract SC-307-04 with Rockwell Collins) under contract MDA904-03-C-1778 with the Department of Defense.

7. REFERENCES

[1] H. P. Barendregt. *The Lambda Calculus.* Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, revised edition, 1991.

[2] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a c compiler front-end. In *Proceedings of Formal Methods*, 2006. Accepted. Available at `http://pauillac.inria.fr/~xleroy/`.

[3] *Common Criteria for Information Technology Security Evaluation (CCITSE)*, Mar. 1999. Available at `http://www.radium.ncsc.mil/tpep/library/ccitse/ccitse.html`.

[4] M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.

[5] Federal Information Processing Standards Publication. Specification for the advanced encryption standard (AES). Technical Report 197, National Institute of Standards and Technology, Nov. 2001. Available at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[6] R. T. C. for Aeronautics (RTCA). DO-178b: Software considerations in airborne systems and equipment certification, Dec. 1992.

[7] A. C. J. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher-Order Logics (TPHOLs)*, pages 25–40, 2003.

[8] D. Greve, R. Richards, and M. Wilding. A summary of intrinsic partitioning verification. In *In Proceedings of the Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2), Austin, TX*, Nov. 2004.

[9] D. S. Hardin, E. W. Smith, and W. D. Young. A Robust Machine Code Proof Framework for Highly Secure Applications. In *Proceedings of the 2006 ACL2 Workshop*, August 2006. Accepted.

[10] T. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[11] S. D. Johnson. *Synthesis of Digital Design from Recursive Equations.* MIT Press, 1984.

[12] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000. ISBN 0792377443.

[13] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer Aided Reasoning: ACL2 Case Studies*, chapter Chapter 8: High-Speed, Analyzable Simulators. Self-Published, Aug. 2002.

[14] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL'06)*, pages 42–54, 2006.

[15] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system: Documentation and user's manual. Available at `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`, 2005.

[16] J. R. Lewis and W. B. Martin. Cryptol: High assurance, retargetable crypto development and validation. In *Proceedings of the IEEE/AFCEA Conference on Military Communications (MILCOM), Boston, MA*, Oct. 2003. Available at `http://www.galois.com/files/milcom.pdf`.

[17] J. Matthews, J. S. Moore, S. Ray, and D. Vroon. Verification condition generation via theorem proving. Submitted, Mar. 2006.

[18] J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[19] J. S. Moore. A grand challenge proposal for formal methods: A verified stack. In *10th Anniversary Colloquium of UNU/IIST*, pages 161–172, 2002.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[21] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, Apr. 1992.

[22] S. L. Peyton-Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003. Available at `http://www.haskell.org/definition/haskell98-report.ps.gz`.

[23] R. L. Rivest1, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The security of the rc6 block cipher. Technical report, RSA Security, 1998.

[24] M. Sheeran. Designing regular array architectures using higher order functions. In J.-P. Jouannaud, editor, *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture (FPCA), Nancy, France*, volume 201 of *LNCS*, pages 220–237. Springer, 1985.

[25] M. Shields. A language for symmetric-key cryptographic algorithms and its implementation. Available at `http://www.cartesianclosed.com/pub/mcryptol/`, Jan. 2006.

[26] M. B. Shields. *μCryptol Reference Manual*, Nov. 2005. Available at `http://www.galois.com/files/mCryptol_refman-0.9.pdf`.

[27] K. Slind, G. Li, and S. Owens. A proof-producing software compiler for a subset of higher order logic. Available at `http://www.cs.utah.edu/~slind/sw-compiler/`, 2006.

[28] K. Slind, S. Owens, J. Iyoda, and M. Gordon. Proof producing synthesis of arithmetic and cryptographic hardware. In *Seventh International Workshop on Designing Correct Circuits DCC: Participants' Proceedings*, 2006. Satellite Event of ETAPS.

[29] D. J. Wheeler and R. M. Needham. TEA, a tiny encryption algorithm. In B. Preneel, editor, *Proceedings of the 1994 Workshop on Fast Software Encryption (FSE), Belgium*, volume 1008 of *LNCS*, pages 363–366. Springer, 1995.