Functional Correctness Proofs of Encryption Algorithms

Jianjun Duan[1], Joe Hurd[2], Guodong Li[1],
Scott Owens[1], Konrad Slind[1], and Junxing Zhang[1]

[1] School of Computing, University of Utah
[2] Oxford University Computer Lab

Abstract. We discuss a collection of mechanized formal proofs of symmetric key block encryption algorithms (AES, MARS, Twofish, RC6, Serpent, IDEA, and TEA), performed in an implementation of higher order logic. For each algorithm, functional correctness, namely that decryption inverts encryption, is formally proved by a simple but effective proof methodology involving application of invertibility lemmas in the course of symbolic evaluation. Block ciphers are then lifted to the encryption of arbitrary datatypes by using modes of operation to encrypt lists of bits produced by a polytypic encoding method.

1 Introduction

Symmetric-key block ciphers represent an important part of today's security infrastructure. Besides their main application, information hiding, block ciphers are also used in the implementation of pseudo-random number generators, message authentication protocols, stream ciphers, and hash functions. There are two main properties that a cipher should have: first, *Functional Correctness*, namely that decryption should invert encryption; second, *Security*, namely that the cipher should be hard to break. In this paper, we focus solely on the first property.

The formal methods community has, to date, paid surprisingly little attention to the functional correctness of block ciphers. This is a pity, since these algorithms provide an application area in which the algorithms are heavily used, security-critical, often well-specified, and well within the scope of theorem proving methods. In this paper, we formalize seven block ciphers and prove their functional correctness.

We have undertaken our proofs in a theorem proving environment: we wanted to see if the seemingly impossible task of brute force analysis of cipher correctness (there would be 2^{128} cases to consider for most of the ciphers we consider) could be avoided by a symbolic analysis. Indeed, it can; we found that the proofs are often quite simple. A major side benefit—which may outweigh the assurance provided by the proofs—is that descriptions of ciphers in higher order logic are elegant and unambiguous. The descriptions are also mathematical and executable. Thus in this work higher order logic is used as a specification language

G. Sutcliffe and A. Voronkov (Eds.): LPAR 2005, LNAI 3835, pp. 519–533, 2005.

for ciphers and its implementation provides a symbolic execution and theorem proving environment. That has two benefits: when prototyping the ciphers, we can use deductive steps to evaluate ciphers on test cases and check results; and we re-use those definitions in the correctness proofs.

In practice, ciphers are used to encrypt compound user-defined data such as numbers, lists, trees, and records. *Modes of operation* [6] can be used to apply a block cipher to the task of encrypting a list of blocks; however, there still remains the issue of how to encrypt higher level datatypes. Often support for this is provided by language-specific libraries. In our work, we have used *polytypism* [11] to implement datatype encryption: elements of datatypes are reduced by polytypic encoders to lists of bits which are then encrypted by a mode of operation instantiated with a particular block cipher. The correctness proofs of block ciphers can be combined with the correctness of encoders to obtain the correctness of data encryption.

This work was initiated in 2002 [17] with a verification of the functional correctness of the then-recent AES standard. We subsequently extended the work to modes of operation, padding, and user-defined datatypes. After that, we were left wondering if the (relative) ease with which AES was verified also held for other block ciphers. Case studies with the other block ciphers mentioned above do seem to indicate that the proof methodology used on AES is widely applicable. The approach (discussed more fully in the sequel) amounts to symbolic evaluation of the formula

$$\forall \mathit{key}\ \mathit{plaintext}.\ \mathsf{decrypt}\ \mathit{key}\ (\mathsf{encrypt}\ \mathit{key}\ \mathit{plaintext}) = \mathit{plaintext}$$

coupled with simplification by *inversion lemmas*, which show that round operations performed during encryption are inverted by their counterparts in decryption. This methodology worked successfully on all our verifications. However, it seems not to be generally automatable: at times, the verification of inversion lemmas can be quite difficult. For example, our proof of invertibility of the column-mixing operation in AES is based on a collection of *ad hoc*, user-specified, lemmas, each proved by brute force. Another example is the verification of IDEA, in which Euclid's extended algorithm needed to be formalized and applied in order to prove invertibility of a special-purpose multiplication operation.

Our verifications[1] have been carried out in HOL-4,[2] an implementation of higher order logic [12]. We have made heavy use of Anthony Fox's HOL-4 library for generating theories and proof tools for fixed-width n-bit words. We use an ML-like functional programming notation to present algorithms in this paper.

2 Encryption Algorithms

Block ciphers usually operate on a fixed, small, amount of data called a *block* which is repeatedly transformed for a number of *rounds*. For example, a block in

[1] Accessible at the webpage `http://www.cs.utah.edu/~slind/papers/lpar05`.
[2] Accessible at the webpage `http://hol.sourceforge.net`.

AES is a 16-tuple of `word8` (8 bit bytes) and each block undergoes ten rounds of transformation. Conceptually, decryption is just 'running encryption in reverse', but often that is not obvious, since decryption round operations can seem quite unrelated to encryption round operations. Although ciphers can have quite complex mathematical underpinnings, their implementations usually require only the most primitive computational objects, found in most machine instruction sets: namely, boolean and arithmetic operations on machine words.

A block cipher takes two inputs: the plaintext and a key. In many cases, before encryption starts, the key is used to generate a *key schedule*, which may be thought of as a list of keys, which get used as encryption proceeds. It is interesting to note that, in many cases, the key schedule calculation is more complex than the actual encryption. We have formalized the computation of key schedules, but have noticed that the actual values in the key schedule are not relevant to functional correctness, at least for the ciphers we have examined. In other words, the key schedule seems to be important for security, and not for functional correctness.

The block ciphers we will examine are *symmetric* key ciphers, which means that the key used for encryption must be used in decryption. Many, but not all, symmetric key ciphers are based on the notion of a *Feistel* network, which divides the plaintext into two halves and repeatedly applies the round function for a number of rounds. In each round, the left half of the plaintext is transformed based on the right half, and then the right half is transformed based on the transformed left half. The round function usually applies several basic linear and non-linear operations: boolean operations such as exclusive-or, substitution (via so-called *S-boxes*), permutation, and modular arithmetic. An S-box is often implemented by an array, but is mathematically just a total function.

In the years previous to 2001, the United States National Institute of Standards (NIST) held a competition to select a successor to the aging DES (Digital Encryption Standard). Among a number of entries, five (MARS, Rijndael, Twofish, Serpent, and RC6) were chosen as finalists, and Rijndael was the eventual winner. Rijndael has since been named AES (Advanced Encryption Standard) and should become widely used in the years ahead.

We have formalized all of the AES finalists, plus a few more ciphers, and proved their functional correctness. In the following, we will introduce each algorithm and discuss any interesting aspects of the correctness proof. Further details on the algorithms can be found in the cited literature.

2.1 AES

The AES block cipher is described in the NIST standards document [13] and in a book [5] by the authors of the cipher. AES is defined for three keylengths: 128, 192, and 256 bits. Our verification is for a keylength of 128, but changing to the other keylengths would be straightforward and involve no changes to the proofs. In the formalization, the original imperative pseudo-code for describing the cipher was converted to a purely functional form which served as an executable model, and also as the code verified in the correctness proof. We followed this

practice for the other ciphers as well. We can define the encryption (`AES`) and decryption (`AES_INV`) functions using function composition as follows:

```
AES keys = from_state ∘ Round 9 (TL keys)
                      ∘ AddRoundKey (HD keys) ∘ to_state

AES_INV keys = from_state ∘ InvRound 9 (TL keys)
                          ∘ AddRoundKey (HD keys) ∘ to_state
```

`AES` takes a key schedule (a list of keys) and `AES_INV` takes the reversed key schedule. The encryptor works by copying the input block into the state, 'xors' the state with the first key, then performs 10 rounds of processing. In each round, one key from the key schedule is consumed. After the rounds of processing are finished, the state is copied to the output. This is formalized as follows: blocks, states, and keys are each represented by 16-tuples of `word8`. The processing of an arbitrary number of rounds is defined by a recursive function named `Round`:

```
(Round 0 [key] state = AddRoundKey key (ShiftRows (SubBytes state))) ∧
(Round (n+1) (key::keys) state =
   Round n keys
    (AddRoundKey key
       (MixColumns
          (ShiftRows (SubBytes state)))))
```

`AddRoundKey` (the names are taken from the original Rijndael documentation) is just pairwise exclusive-or; `SubBytes` applies an S-box to each element of the state; and `ShiftRows` performs a simple permutation on the block. The most complex operation is `MixColumns`; it treats the state as a 4×4 matrix and applies a specialized transformation on each column of the matrix. Mathematically, each column in the state is treated as a four-term polynomial over $\mathbf{GF}(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial. When encrypting, the fixed polynomial is $a(x) = \mathbf{03}x^3 + \mathbf{01}x^2 + \mathbf{01}x + \mathbf{02}$, while decryption uses the polynomial $a^{-1}(x) = \mathbf{0B}x^3 + \mathbf{0D}x^2 + \mathbf{09}x + \mathbf{0E}$. In the implementation, column multiplication during encryption is implemented by

```
MultCol (a,b,c,d) =
  ((02 • a) ⊕ (03 • b) ⊕ c ⊕ d,      (* F1 *)
    a ⊕ (02 • b) ⊕ (03 • c) ⊕ d,     (* F2 *)
    a ⊕  b  ⊕ (02 • c) ⊕ (03 • d),   (* F3 *)
   (03 • a) ⊕  b ⊕  c ⊕ (02 • d))    (* F4 *)
```

where $(- \bullet -)$ is the finite field multiplication and $\oplus$ is exclusive-or. The actual appplication of `MultCol` to the block is by the following function (where we have arranged the state tuple so as to suggest a matrix):

```
MixColumns (b1, b2, b3, b4,
            b5, b6, b7, b8,
            b9, b10,b11,b12,
            b13,b14,b15,b16) =
 let (b1',b5',b9',b13')  = MultCol (b1,b5,b9,b13)
 and (b2',b6',b10',b14') = MultCol (b2,b6,b10,b14)
 and (b3',b7',b11',b15') = MultCol (b3,b7,b11,b15)
```

```
  and (b4',b8',b12',b16') = MultCol (b4,b8,b12,b16)
  in
    (b1',b2', b3',b4',b5',b6',b7',b8',
     b9',b10',b11',b12',b13',b14',b15',b16')
```

Decryption also uses `MixColumns`, except that `MultCol` has been replaced by `InvMultCol`:

```
InvMultCol (a,b,c,d) =
  ((0E • a) ⊕ (0B • b) ⊕ (0D • c) ⊕ (09 • d),  (* G1 *)
   (09 • a) ⊕ (0E • b) ⊕ (0B • c) ⊕ (0D • d),  (* G2 *)
   (0D • a) ⊕ (09 • b) ⊕ (0E • c) ⊕ (0B • d),  (* G3 *)
   (0B • a) ⊕ (0D • b) ⊕ (09 • c) ⊕ (0E • d))  (* G4 *)
```

Verification. Functional correctness, namely

```
∀keys block. INV_AES (reverse keys) (AES keys block) = block
```

is proved as follows: the variable `block` is split into a 16-tuple of `word8` variables. Then all the definitions used to define `AES` and `AES_INV` are expanded by symbolic evaluation. This results in (conceptually) a long string of function compositions:

```
from_state ∘ .... ∘ to_state ∘ from_state ∘ ... ∘ to_state
```

and then we need merely simplify with inversion lemmas, showing that each operation used in encryption inverts its counterpart in decryption. Most of the inversion lemmas for AES are quite easy to prove: `from_state` inverts `to_state` (and *vice versa*), $\oplus$ inverts itself, the S-boxes are inverses, when regarded as functions, and so on. However, the inversion lemma for column mixing

```
InvMixColumns (MixColumns s) = s
```

is difficult to prove. Naive attempts at this led to overly large goals, and we were forced to much more basic steps. To see the problem, let us consider the action on a column (a, b, c, d). In the forward direction, `MixColumns` applies transformations $F_1 \cdots F_4$ to the column

$$\begin{aligned} a' &= F_1(a, b, c, d) \\ b' &= F_2(a, b, c, d) \\ c' &= F_3(a, b, c, d) \\ d' &= F_4(a, b, c, d) \end{aligned}$$

and in the reverse `InvMixColumns` applies transformations $G_1 \cdots G_4$ to the resulting column

$$\begin{aligned} a'' &= G_1(a', b', c', d') \\ b'' &= G_2(a', b', c', d') \\ c'' &= G_3(a', b', c', d') \\ d'' &= G_4(a', b', c', d') \end{aligned}$$

and we then wish to show that $a = a'', b = b'', c = c'', d = d''$. Consideration of a should illustrate the strategy.

$$
\begin{aligned}
a' &= (\mathbf{02} \bullet a) \ \oplus \ (\mathbf{3} \bullet b) \ \oplus \ c \ \oplus \ d \\
b' &= a \ \oplus \ (\mathbf{02} \bullet b) \ \oplus \ (\mathbf{03} \bullet c) \ \oplus \ d \\
c' &= a \ \oplus \ b \ \oplus \ (\mathbf{02} \bullet c) \ \oplus \ (\mathbf{03} \bullet d) \\
d' &= (\mathbf{03} \bullet a) \ \oplus \ b \ \oplus \ c \ \oplus \ (\mathbf{02} \bullet d)
\end{aligned}
$$

Thus a'' is

$$(\mathbf{0E} \bullet a') \ \oplus \ (\mathbf{0B} \bullet b') \ \oplus \ (\mathbf{0D} \bullet c') \ \oplus \ (\mathbf{09} \bullet d')$$

which expands to

$$
\begin{aligned}
&(\mathbf{0E} \bullet ((\mathbf{02} \bullet a) \ \oplus \ (\mathbf{03} \bullet b) \ \oplus \ c \ \oplus \ d)) \oplus \\
&(\mathbf{0B} \bullet (a \ \oplus \ (\mathbf{02} \bullet b) \ \oplus \ (\mathbf{03} \bullet c) \ \oplus \ d)) \oplus \\
&(\mathbf{0D} \bullet (a \ \oplus \ b \ \oplus \ (\mathbf{02} \bullet c) \ \oplus \ (\mathbf{03} \bullet d))) \oplus \\
&(\mathbf{09} \bullet ((\mathbf{03} \bullet a) \ \oplus \ b \ \oplus \ c \ \oplus \ (\mathbf{02} \bullet d)))
\end{aligned}
$$

By use of associativity and commutativity of $\oplus$ and distribution of $\bullet$ over $\oplus$, we can separate the expression into four subexpressions each involving only one variable. Each subexpression is then simplified by case analysis on the 256 ways of forming a `word8` quantity. The subexpression involving a is simplified to a, and the subexpressions for b, c, d all simplify to $\mathbf{0}$, leading to the conclusion $a'' = a$. Such a proof was carried out for each of a, b, c, d. The potential tedium of this was eased by HOL's rewriter, which can perform permutative rewriting (in this case using the associativity and commutativity of $\oplus$).

Enhancements and Optimizations. Working in a theorem prover means that program transformations and optimizations can be easily applied, once proved. For example, in [17], an optimization to the decryption process is verified, and the resulting decryptor is proved to be mathematically equal to the original. As another example, the multiplication used in AES may be specified recursively, iteratively, or as a lookup table (feasible since all multiplications have one argument fixed to one of a small set of constants). In our development, we prove the iterative and recursive functions equal, and generate the tables by proof from the recursive algorithm, achieving high assurance. Thus multiple implementations can be spawned, by proof, from a single source.

In summary, the functional correctness of AES was straightforward, except for one lemma, which required ingenuity in the decomposition. One question we have is whether the difficulty of the proof of invertibility of column mixing is instrinsic, or whether it would be easier as a general argument at the level of finite fields and polynomials. It may also be a good challenge for SAT methods.

2.2 Verifying the Other Ciphers

We now discuss the other ciphers, omitting much detail since the basic ideas have been established in the discussion of AES.

MARS. [4] was IBM's candidate in the AES competition. It has 128 bit blocks (a 4-tuple of `word32`) and a variable keysize ranging from 128 to 448 bits (we

chose 128). The key schedule is a 40-tuple of `word32`s. Encryption in MARS takes place in three phases: eight rounds of forward mixing, sixteen rounds in the *cryptographic core*, and eight rounds of backwards mixing. Decryption applies counterparts of these three phases. MARS uses a 512 element S-box. Although the formalization of MARS is quite complex, the basic operations are simple boolean operations and addition plus application of the S-boxes. Inversion lemmas for mixing and the cryptographic core are quite easy; again symbolic evaluation plus rewriting with inversion lemmas and some basic word identities (algebraic properties of exclusive-or, for example) sufficed for the final theorem.

Twofish. [16] was also an AES competitor. It has a block size of 128 bits and key sizes up to 256 bits. Twofish's distinctive features are the use of pre-computed key-dependent S-boxes, and a relatively complex key schedule. Twofish is a 16-round Feistel network. We used `word4`, `word8`, and `word32` in the formalization. A block is a 4-tuple of `word32`, and the key schedule is a 40-tuple of `word32`, computed from 32 `word8`s. Twofish uses several multiplication operations, which are similar to that of AES. It uses these in column multiplication, also much like that of AES. However, unlike AES, the correctness proof for Twofish is almost comically easy.

RC6. [15] is a block cipher based on RC5 and designed by Rivest, Sidney, and Yin for RSA Security. RC6 is a parameterized algorithm where the block size, the key size, and the number of rounds are variable; the upper limit on the key size is 2040 bits. In our formalization, we have fixed on a internal block size of 176 bits (a 6-tuple of `word32`), a key size of 64 (a pair of `word32`), and twenty rounds. RC6 uses integer multiplication to increase the diffusion achieved per round so that fewer rounds are needed and the speed of the cipher can be increased. The algorithm also wraps the round computations in 'pre-whitening' and 'post-whitening' steps. RC6 does not use S-boxes. In spite of the fact that multiplication is used, the verification of RC6 was extremely simple, reducing to simple identities on words.

TEA. (Tiny Encryption Algorithm) [22] is a very compact cipher designed by David Wheeler and Roger Needham. TEA operates on 64-bit blocks and uses a 128-bit key. TEA has has a trivial key schedule (the same four keys are used throughout); it also does not use an S-box. It has Feistel structure, using addition and subtraction as the reversible operators rather than exclusive-or. A dual shift causes all bits of the key and data to be mixed repeatedly. The number of rounds before a single bit change of the data or key has spread very close to 32 is at most six, so that sixteen cycles may suffice and the authors suggest 32 (we implemented 32 rounds). The verification of **TEA** was again an easy application of our methodology.

Serpent. [1] is a 128-bit block cipher designed by Ross Anderson, Eli Biham and Lars Knudsen. It placed second in the AES competition. The authors designed Serpent to provide users with the highest practical level of assurance that no shortcut attack will be found. To achieve this, the cipher uses twice as many rounds (32) as are sufficient to block all currently known shortcut

attacks. Despite this intentional 'overdesign', Serpent supports a very efficient bitslice implementation. We have verified both the bitslice implementation and a more conventional reference implementation. Perhaps surprisingly, the bitslice implementation was far easier to verify than the reference version! The reference implementation of Serpent uses tables for S-boxes, linear transformations, and permutations. The specification used lists of indices, and we had to derive functions, which were more tractable in later proofs, from them. Several transcription errors were caught in the later invertibility proofs.

IDEA. [8] is used in the popular PGP (Pretty Good Privacy) package. IDEA operates on 64-bit blocks using a 128-bit key, and consists of seventeen rounds. The processes for encryption and decryption are similar. IDEA derives much of its security by interleaving operations from different algebraic groups: exclusive-or, addition modulo 2^{16}, and multiplication modulo $2^{16}+1$ (a prime), where 0 is treated as 2^{16}. The internal operations of IDEA use `word16`, so the state is a 4-tuple of `word16` and the input key is treated as an 8-tuple of `word16`. The key schedule is a 52-tuple of `word16`.

The verification of IDEA is straightforward, much like the others, except for proving the invertibility of the multiplication. The difficulty comes from the fact that the native multiplication in `word16` is modulo 2^{16}, and not modulo $2^{16}+1$. So we had to define our own multiplication and give an implementation for its inverse. This required some new formalization work: we had to define the generalized Euclid's algorithm, develop relevant properties, and show that the algorithm does find multiplicative inverses modulo 65537 for all numbers from 1 to 65536 inclusive. A further complication is that, since 65537 can not be represented in 16 bits, we had to map multiplications out to a larger type, and then map back. A full account is given in [23].

In summary, the verification of IDEA has much in similarity with that of AES: mostly the proof was easy, except for one complex operation.

3 Data Encryption

We now turn from block ciphers to techniques for encrypting data. The first step is to formalize so-called modes of operation. A *mode of operation* extends a cipher from single blocks to arbitrary block sequences. Some acronyms of the commonly used modes are ECB, CBC, CFB, OFB, and CTR [6]. In this paper, we chose to work with CBC (Cipher Block Chaining). In CBC, the previous ciphertext block is 'xor'ed with the current plaintext before encryption. We formalize CBC (see Fig. 1) as a pair of recursive functions being parameterized by block encryptors and decryptors *enc* and *dec*. The parameter *xor* represents an 'xor' function: since we do not know *a priori* what the actual type of blocks will be (different ciphers have different representations for blocks), we simply fill *xor* in later.[3] In the actual formalization, `block` is just a type variable in the HOL logic, to be instantiated to the block type of a particular cipher.

[3] A logic with dependent types could avoid this extra parameterization.

$$
\begin{array}{l}
\mathsf{CBC}\ xor\ enc\ \_\ [\,] \quad\quad = [\,] : \mathsf{block\ list} \\
\mathsf{CBC}\ xor\ enc\ v\ (h :: t) \quad = \mathtt{let}\ x = enc\ (xor\ h\ v)\ \mathtt{in}\ x :: \mathsf{CBC}\ xor\ enc\ x\ t \\
\\
\mathsf{CBC}^{-1}\ xor\ dec\ \_\ [\,] \quad\quad = [\,] : \mathsf{block\ list} \\
\mathsf{CBC}^{-1}\ xor\ dec\ v\ (h :: t) = xor\ (dec\ h)\ v :: \mathsf{CBC}^{-1}\ xor\ dec\ h\ t
\end{array}
$$

Fig. 1. Cipher Block Chaining

Both CBC and CBC^{-1} have the type

$$(\mathsf{block} \to \mathsf{block} \to \mathsf{block}) \to (\mathsf{block} \to \mathsf{block}) \to \mathsf{block} \to \mathsf{block\ list} \to \mathsf{block\ list}\ .$$

The correctness of CBC using arbitrary inverting encryptors and decryptors

$$
\begin{array}{l}
\vdash \forall \ell\ xor\ v\ encrypt\ decrypt. \\
\quad (decrypt \circ encrypt) = \mathsf{I} \wedge \\
\quad (\forall x\ y.\ (x\ xor\ y)\ xor\ y = x) \\
\quad\quad \Rightarrow \\
\quad \forall k.\ \mathsf{CBC}^{-1}\ (xor)\ (decrypt\ k)\ v\ (\mathsf{CBC}\ (xor)\ (encrypt\ k)\ v\ \ell) = \ell
\end{array}
$$

is proved very easily by induction on ℓ. From there, support for data encryption is provided by adding in functions for encoding and decoding arbitrary data, as can be seen in the following trivial consequence:

$$
\begin{array}{l}
\vdash \forall (encode : \alpha \to \mathsf{bool\ list})\ (decode : \mathsf{bool\ list} \to \alpha) \\
\quad (block : \mathsf{bool\ list} \to \mathsf{block\ list})\ (unblock : \mathsf{block\ list} \to \mathsf{bool\ list}) \\
\quad (encrypt : \mathsf{block} \to \mathsf{block})\ (decrypt : \mathsf{block} \to \mathsf{block})\ xor. \\
\quad\quad (decode \circ encode = \mathsf{I}) \wedge \\
\quad\quad (\forall k.\ decrypt\ k \circ encrypt\ k = \mathsf{I}) \wedge \\
\quad\quad (unblock \circ block = \mathsf{I}) \wedge \\
\quad\quad (\forall x\ y.\ (x\ xor\ y)\ xor\ y = x) \\
\quad\quad \Rightarrow \forall v\ key. \\
\quad\quad\quad\quad (decode \circ unblock \circ (\mathsf{CBC}^{-1}\ (xor)\ (decrypt\ key)\ v)) \circ \\
\quad\quad\quad\quad (\mathsf{CBC}\ (xor)\ (encrypt\ key)\ v \circ block \circ encode) = \mathsf{I}
\end{array}
\tag{1}
$$

In other words, provided that inverting encoder/decoder, blocker/unblocker, and encryptor/decryptor are provided, then (a) encoding the data to a list of bits then (b) chunking the list into blocks then (c) using CBC to encrypt the blocks can be inverted by applying the inverse operations in the correct order. Note that there is a hidden complexity, in that the action of turning a list of bits into a list of fixed length blocks requires *padding* the bits to get a list the length of which is a multiple of the block size. Moreover, padding must be implemented in such a way that the extra padding can be dropped off when mapping back from a block to a list of bits.

We have now finished the abstract development. To see how it may be instantiated, we turn our attention to type-directed construction of encoders and decoders.

3.1 Encoding and Decoding Datatypes

A common task in computer science is to package up high-level data as flat strings of bits, and correspondingly, to unpack strings of bits in order to recover the high-level data. When this is done to send data over a communication network, it is called marshalling or serialization but we will use *encoding/decoding* or simply *coding*. A type-directed approach to coding, based on an interpretation of higher order logic types into higher order logic terms, is given in [18]. An encoding function can be thought of simply as an injective function of type $\tau \to \mathsf{bool\ list}$ mapping elements of type τ to lists of booleans. The injectivity condition prevents two elements of τ being encoded as the same list of booleans, and so guarantees that if a list can be decoded then the decoding will be unique.

Encoding functions can be automatically defined when a new datatype is declared; the interpretation is used to calculate the form of the encoder from the declaration of the type. Mutually recursive datatypes and datatypes with recursion under existing type operators (so-called *nested* datatypes) are cleanly handled. Encoding and decoding of polymorphic types is dealt with by abstraction: an encoder for a polymorphic type is parameterized by encoders for types that may be substituted for the type variables.

Without going into the details of encoding, which may be found in [18], each constructor for a datatype is assigned a *marker list*, which serves to distinguish it from other constructors for the type. Lists have two constructors, and so the marker lists have length one. A datatype with eight constructors would need marker lists of length three.

For example, the encoding function for the datatype $\alpha\ \mathsf{list}$ of polymorphic lists is the following:

$$\begin{aligned} \mathsf{encode\_list}\ f\ [\,] &\equiv [\mathsf{F}] \wedge \\ \mathsf{encode\_list}\ f\ (h :: t) &\equiv \mathsf{T} :: f\ h\ @\ \mathsf{encode\_list}\ f\ t \end{aligned}$$

where $f : \alpha \to \mathsf{bool\ list}$ is the parameter encoder. Lists have two constructors, which are distinguished by the prepending of marker lists $[\mathsf{F}]$ and $[\mathsf{T}]$.

Although encoders are automatically defined for every datatype declared in HOL, a user may wish to override the automatic definition with an alternative version, or to provide an encoder for a non-datatype *e.g.*, for finite sets. A custom encoder for natural numbers[4] is the following:

$$\begin{aligned} \mathsf{encode\_num}\ n \equiv\ & \mathtt{if}\ n = 0\ \mathtt{then}\ [\mathsf{T};\ \mathsf{T}] \\ & \mathtt{else\ if}\ \mathsf{even}\ n\ \mathtt{then}\ \mathsf{F} :: \mathsf{encode\_num}\ ((n-2)\ \mathsf{div}\ 2) \\ & \mathtt{else}\ \mathsf{T} :: \mathsf{F} :: \mathsf{encode\_num}\ ((n-1)\ \mathsf{div}\ 2) \end{aligned}$$

A typical environment for encoding functions would include at least the following bindings:

[4] Built up from 0 using two successor functions: $2n + 1$ and $2n + 2$.

type	encoder
$\tau_1 * \tau_2$	encode_prod $f\ g\ (x, y) \equiv f\ x\ @\ g\ y$
$\tau_1 + \tau_2$	encode_sum $f\ g$ (INL x) $\equiv$ F :: $f\ x$
	encode_sum $f\ g$ (INR y) $\equiv$ T :: $g\ y$
bool	encode_bool $x \equiv [x]$
option	encode_option f NONE $\equiv$ [F]
	encode_option f (SOME x) $\equiv$ T :: $f\ x$
num	encode_num (defined above)
τ list	encode_list (defined above)

Encrypting Data. For an example, we will see how to synthesize encryption routines for the type (num∗bool option)list. Given a typical encoder environment, traversing the type structure and emitting the corresponding encoders yields the following function in the HOL logic:

```
encode_list (encode_prod encode_num (encode_option encode_bool))
```

Applying it to the list `[(1,NONE); (13,SOME T); (257,SOME F)]` yields the theorem

```
|- encode_list (encode_prod encode_num (encode_option encode_bool))
       [(1,NONE); (13,SOME T); (257,SOME F)]
   = [T; T; F; T; T; F; T; T; F; F; F; T; T; T; T; T; T; F;
      F; T; F; T; F; T; F; T; F; T; F; T; F; T; T; T; F; F]
```

If we instantiate `CBC` with the TEA block cipher, the key `(1w,2w,3w,4w)` and the initial value `(5w,10w)` for v, and prepend encoding, padding, and blocking, we can deductively evaluate the expression to obtain a theorem giving the result of encrypting our specific input list:

```
|- (CBC XORB (TEAEncrypt (1w,2w,3w,4w)) (5w,10w) o BLOCK o PAD o
     encode_list (encode_prod encode_num (encode_option encode_bool)))
     [(1,NONE); (13,SOME T); (257,SOME F)]
   = [(3008902428w,1274536877w)]
```

Decrypting Data. A decoder for type τ is an algorithm that takes as input a list of booleans and returns an element of type τ. It is also possible to build and compose decoders in a type-directed way. The key is to think of a decoder for type τ as a monadic parser [21] :

$$\mathsf{decode\_}\tau : \mathsf{bool\ list} \rightarrow (\tau \times \mathsf{bool\ list})\ \mathsf{option}$$

Such a function tries to parse an input list of booleans into an element of type τ, and if it succeeds then it returns the element of τ, together with the list of booleans that were left over. If it fails to parse the input list, it signals this by returning NONE. (A decoding function of the expected type bool list $\rightarrow \tau$ can be easily recovered when decoding is expected to succeed.) As an example, the following is the decoding function for lists:

```
wf_decoder d ⇒
  (decode_list d [] = NONE)  ∧
  (decode_list d (F::t) = SOME ([],t))   ∧
  (decode_list d (T::t) =
     case d t
       of NONE -> NONE
       || SOME (x, t') ->
            case decode_list d t'
             of NONE -> NONE
             || SOME (xs, t'') -> SOME (h::xs, t''))
```

Thus, given a parameter decoder d, one decodes to an empty list, provided the marker at the head of the bits list is `F`; otherwise, the marker must be `T`, and we expect to be able to use d to deliver the head of the original list x and remaining bits t'. We then recurse to get the rest of the original list xs, and the remaining bits t''. As HOL is a logic of total functions, this function is only well-defined if d does not increase the length of the list of bits; this is enforced by the constraint wf_decoder d.

In our current formalization, a decoding function also has an attached *domain predicate*, in order to deal with subsets of types. We have omitted the domain predicates since they hamper readability, and are not actually used in a significant way in our experiments so far.

Returning to our example, suppose we have a decoder context containing at least decoders for the types num, list, option, and bool, then a type-directed traversal of (num ∗ bool option) list yields the following decoding function.

```
decode_list (decode_prod (decode_num (decode_option decode_bool)))
```

In order to formally apply the abstract inversion theorem (1) for data encryption, we need to show that the synthesized decoder inverts the synthesized encoder. This is relatively easy to automate by backchaining with pre-proved theorems relating basic coders/decoders already in the coder and decoder contexts. Thus we ultimately have that

```
(decode_list (decode_prod (decode_num (decode_option decode_bool))) o
 UNPAD o UNBLOCK o CBC_DEC XORB (TEADecrypt (1w,2w,3w,4w)) (5w,10w))
o
(CBC XORB (TEAEncrypt (1w,2w,3w,4w)) (5w,10w) o BLOCK o PAD o
 encode_list (encode_prod encode_num (encode_option encode_bool)))
```

is the identity function. In summary, compound encoders and decoders can be formally synthesized and their invertibility property proved in the theorem prover; this property can then be used to show that data encryption for the specified type is invertible.

4 Related Work

Probably the earliest application of a proof assistant to cryptography is the use of Boyer and Moore's Nqthm to verify the invertibilty of encryption in the

RSA public-key algorithm [14]. Whereas their goal seemed to be to check an interesting piece of (then) recently-announced mathematics, we have been more interested in getting an overview of how hard proofs are for a gamut of algorithms in this area.

In [17] we verified the functional correctness of Rijndael, and in [23] we provide further detail on the functional correctness of the IDEA cipher. Toma and Borrione report on an ACL2 verification of an implementation of the SHA-1 hash algorithm in [20]. Higher level security protocol specification and verification has received much more attention than ciphers, and this work is starting to mature: see [2] for example. It would be interesting to explore links between our correctness proofs and that body of work. Finally, the Cryptol language [9] is a domain-specific language, based on functional programming principles, aimed at cryptographers. Cryptol provides a uniform stream-based view of all the data involving in encryption, and supports that view with an interesting type system reflecting how functions manipulate streams. C code can be generated from Cryptol programs, and there is also a path to FPGAs.

5 Conclusions and Further Work

This paper summarizes some case studies in the verification of block ciphers formalized in higher order logic. A simple proof methodology successfully supports functional correctness proofs of these algorithms. Although some ciphers are formulated in terms of concepts from abstract algebra and number theory, we found that in most cases (IDEA was the sole exception) higher mathematics could be avoided in the proofs. We also showed how ciphers can be lifted from blocks to arbitrary user-defined datatypes by use of modes of operation and polytypic encoding techniques.

This activity takes place inside the theorem prover, and although it is encouraging to see that bespoke data encryption can be supported in such an environment, it would be a useful next step to *generate* executable models in real programming languages from our formal models. In fact, we can already do that in HOL-4, generating standalone ML code from the formal specifications. In principle, the generated code could be compiled in with other code to build an application with a formally-justified security component. It would also be useful to input or output code in mainstream languages such as C or Java, as a way of developing a path from verification environments to security applications development. The paper [3] appears to provide an interesting framework in which to work.

We have also been investigating the automatic synthesis of hardware from our specifications using a prototype deduction-based compiler [7]. At present, we are able to generate netlists from the HOL-4 specification of AES, and we plan to further develop and test our prototype on the other ciphers presented here.

Invertibility proofs, as we have seen, are in many cases quite straightforward. It would therefore be interesting to see how much of these proofs could be automated. However, as in AES and IDEA, there can be round operations that have

hard to prove inversion lemmas, but that itself is interesting information about a cipher.

An interesting point is that key schedule generation algorithms can be more difficult than the actual encryption core. This means that mistakes can be more easily made when implementing them. In our work, we formalized these algorithms, but proved little about them, other than a few facts about how long the resulting schedule would be, for example. Therefore, correctness properties of key schedules, if such exist and are amenable to mechanized formal proof, could lead to even higher levels of assurance.

We are currently investigating links between HOL-4 and Cryptol. Since Cryptol is a stream-processing language, and its semantics document is not yet in the public domain, we are basing the work on a HOL theory of lazy lists, due to Michael Norrish (based on original work by John Matthews [10]). Several of the ciphers have been ported to work over the new type, and we have been encouraged, since the functional correctness proof of the new algorithm can be reduced with a few simple lemmas to that of the old. A longer-term goal would be to provide a HOL shallow embedding of an interesting subset of Cryptol.

Another operation sometimes used with encryption is compression. It would be interesting to incorporate a formally verified compression algorithm. Since compression, being invertible, is similar to encryption, there may be commonalities in the two formal exercises. A verification of Huffman's algorithm has recently been carried out in the Coq system [19], and there are many other important compression algorithms that could be tackled.

Finally, the investigation of security properties of block ciphers in theorem provers seems to be an obvious area for future work.

References

1. R. Anderson, E. Biham, and L. Knudsen, *Serpent: A proposal for the advanced encryption standard*, Available at http://www.cl.cam.ac.uk/~rja4/serpent.html, August 1998.
2. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.C. He, M. Rusinowitch, J. Santiago, M. Turuani, L. Vigano, and L. Vigneron, *The Avispa tool for the automated validation of internet security protocols and applications*, Proceedings of Computer Aided Verification (CAV), Springer LNCS, no. 3576, 2005.
3. Michael Backes, Birgit Pfitzmann, and Michael Waidner, *Symmetric authentication in a simulatable Dolev-Yao style cryptographic library*, Journal of Information Security **4** (2005), no. 3, 135–154.
4. C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. Mathas Jr., L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, *MARS - a candidate cipher for AES*, Available at http://www.research.ibm.com/security/mars.pdf, September 1999.
5. Joan Daemen and Vincent Rijmen, *The design of Rijndael: AES - the Advanced Encryption Standard*, Information Security and Cryptography, no. 17, Springer-Verlag, 2002.

6. Morris Dworkin, *Recommendation for block cipher modes of operation: Methods and techniques*, Tech. Report SP 800-38A, National Institute of Standards and Technology, 2001.
7. M. Gordon, J. Iyoda, S. Owens, and K. Slind, *Automatic formal synthesis of hardware from higher order logic*, Proceedings of Fifth International Workshop on Automated Verification of Critical Systems (AVoCS), ENTCS, 2005, to appear.
8. X. Lai, J.L. Massey, and S. Murphy, *Markov ciphers and differential cryptanalysis*, Advances in Cryptology - Eurocrypt '91 (Donald W. Davies, ed.), LNCS, vol. 547, Springer Verlag, 1991, pp. 17–38.
9. Jeff Lewis, *Cryptol, a domain specific language for cryptography*, Tech. report, Galois Connections Inc., 2002, URL—http://www.cryptol.net/docs/CryptolPaper.pdf.
10. John Matthews, *Recursive definition over coinductive types*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999.
11. Lambert Meertens, *Calculate polytypically!*, Proc. 8th Int. Symp. on Programming Languages: Implementations, Logics, and Programs, PLILP'96 (H. Kuchen and S.D. Swiestra, eds.), vol. 1140, Springer-Verlag, Berlin, 1996, pp. 1–16.
12. Michael Norrish and Konrad Slind, *HOL-4 manuals*, 1998-2005, Available at `http://hol.sourceforge.net/`.
13. United States National Institute of Standards and Technology, *Advanced Encryption Standard*, Web: http://csrc.nist.gov/encryption/aes/, 2001.
14. R.Boyer and J Moore, *Proof checking the RSA public key encryption algorithm*, American Mathematical Monthly **91** (1984), no. 3, 181–189.
15. R. Rivest, M. Robshae, R. Sidney, and Y.L. Yin, *The RC6 block cipher*, Available at http://www.rsasecurity.com/rsalabs/rc6, August 1998.
16. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, *The Twofish encryption algorithm*, John Wiley and Sons, 2003.
17. Konrad Slind, *A verification of Rijndael in HOL*, Supplementary Proceedings of TPHOLs 2002 (V. A Carreno, C. A. Munoz, and S. Tahar, eds.), NASA Conference Proceedings, no. CP-2002-211736, August 2002.
18. Konrad Slind and Joe Hurd, *Applications of polytypism in theorem proving*, Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rome, Italy, Proceedings (D. Basin and B. Wolff, eds.), Lecture Notes in Computer Science, vol. 2758, Springer, September 2003, pp. 103–119.
19. Laurent Thery, *Formalizing Huffman's algorithm*, Tech. Report TRCS 034/2004, Department of Informatics, University of Acquila, 2004.
20. Diana Toma and Dominique Borrione, *Formal verification of a SHA-1 circuit core using ACL2*, Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs'05 (Joe Hurd and Tom Melham, eds.), Lecture Notes in Computer Science, vol. 3603, Springer-Verlag, August 2005, pp. 326–341.
21. Phil Wadler, *Monads for functional programming*, Marktoberdorf Summer School on Progam Design Calculi (M. Broy, ed.), NATO ASI Series F: Computer and Systems Sciences, vol. 118, Springer-Verlag, 1992.
22. David Wheeler and Roger Needham, *TEA, a tiny encryption algorithm*, Fast Software Encryption: Second International Workshop, Lecture Notes in Computer Science, vol. 1008, Springer Verlag, 1999, pp. 363–366.
23. Junxing Zhang and Konrad Slind, *Verification of Euclid's algorithm for finding multiplicative inverses*, Emerging Trends: Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005 (Joe Hurd, ed.), 2005.