Proving AES Correctness in Coq

Paras Chetal, Ananth Dandibhotla, Rayan Kashghari
Georgia Institute of Technology
{pchetal3, ananth.dandibhotla, rkashghari3}@gatech.edu

***Abstract*—Advanced Encryption Standard (AES) is a National Institute of Standards and Technology (NIST) standardized block cipher for symmetric encryption. AES has a block size of 128 bits and includes three key configurations of 128, 192 and 256 bit key-lengths. In this project we prove the cryptographic correctness property of the AES-128 block cipher in the Electronic Code Book (ECB) and Cipher Block Chaining (CBC) modes of operations: for any 128-bit key and for any plaintext, decryption under the key of a ciphertext which was the result of encryption of the plaintext under the same key gives back the original plaintext. We start from scratch and define the types and functions for AES-128 encryption and the corresponding inverse functions for decryption. We prove correctness for functions and their inverses, and build our proof for AES-128 correctness using these sub-proofs. We then prove the correctness of the ECB and CBC mode-of-operations using AES-128 as a building block. We prove this for any arbitrary length plaintext using induction on an infinite type, and using the AES-128 correctness proof. Finally, we extract usable proved-correct encryption and decryption code for OCaml, Haskell and Scheme.**

I. Introduction

Blockciphers are the critical building blocks upon which shared-key symmetric encryption schemes are built. A blockcipher is a permutation $E : \{0,1\}^k \rightarrow \{0,1\}^n \rightarrow \{0,1\}^n$ that takes as input a k-bit key and an n-bit plaintext block and outputs an n-bit ciphertext block. The inverse function $E^{-1} : \{0,1\}^k \rightarrow \{0,1\}^n \rightarrow \{0,1\}^n$ does the opposite.

The NIST standard for blockciphers: AES-128 takes as input a 128 bit key to instantiate an encryption function that maps a 128 bit plaintext to a 128 bit ciphertext. The AES-128 decryption function similarly takes in a 128-bit key and maps a 128 bit ciphertext to a 128 bit plaintext. Instantiated with the same key, both encryption and decryption are deterministic functions which parse the 128-bit block as a 16 byte 4x4 state matrix and perform invertible transformations on it to get the 128-bit block output. These invertible transformations are done as parts of different rounds within encryption and decryption. Unfolding encryption and decryption and their rounds thus leads to the inverse transformations cancelling out the effect of the transformations and output the original input block.

To prove this in Coq we first implement the AES-128 blockcipher encryption and decryption functions by defining the types and transformations underlying the block cipher. The proof for correctness in Coq takes a similar shape where we first prove sub-proofs involving the invertible transformations: that the inverse transformations reverse the effect of the transformations and return the state matrix to its original state. We string together these sub-proofs to prove AES-128 correct.

While blockciphers allow us to encrypt or decrypt messages from the message space of all 128 bit strings, practically we are required to encrypt messages of arbitrary length. This is where symmetric encryption schemes come in.

A symmetric encryption scheme $SE = (KeySpace, E, D)$ is defined for messages from the set $MessageSpace$ using a set of keys $KeySpace$ (or equivalently, a randomized key generation algorithm K), an encryption algorithm E and a decryption algorithm D. While encryption may be randomized or stateful (and ideally should not be deterministic for security), decryption is required to be deterministic. The cryptographic correctness property of symmetric encryption is thus: $\forall k \epsilon KeySpace, \forall m \epsilon MessageSpace, D_k(E_k(m)) = m$. Practically, $MessageSpace$ is set of arbitrary length bit-strings $\{0,1\}^*$ and $KeySpace$ is a set of fixed-length bit-strings.

In practice, symmetric encryption schemes are generally built by splitting a message into fixed-size blocks and using a blockcipher in different modes-of-operation. The most straight-forward mode-of-operation is to run each of these plaintext blocks through the blockcipher initialized with the key and output a concatenation of the resulting ciphertext blocks. Decryption does the same but using the inverse of blockcipher. This is the Electronic Code Book (ECB) mode of operation. However, this is deterministic and therefore insecure as it leaks information: same plaintexts get encrypted to same ciphertexts. A much more secure (formally, IND-CPA indistinguishable under chosen plaintext attacks) mode-of-operation is the randomized Cipher Block Chaining mode (CBC) which uses a random initialization vector and chains together outputs of each blockcipher using XORs.

To prove correctness of AES-128-ECB and AES-128-CBC, we first implement the algorithms themselves by defining a recursive type (blocks), the recursive functions for encryption and decryption using Coq's Fixpoint notation, and the AES-128 implementation from before. We then prove correctness using induction on the block type, proving correctness for each mode-of-operation by first proving correctness for a single block and then proving correctness for a message with an additional block using the generated correctness hypothesis

for a previous block-size message.

II. Motivation

Most secure block ciphers apply a complex set of transformation to the plaintext in order to generate the ciphertext. The main goal is to create a cipher text that doesn't leak any information of the original message that was encrypted, such that an adversary with reasonable resources is unable to invert the cipher text, as long as long as the key remains secret. While block ciphers such as AES are conjectured to be PRF (pseudo-random-function) secure, real world implementations often introduce vulnerabilities making cryptographic primitives involving them insecure. Several vulnerabilities that arise in cryptography actually result from the implementation of the algorithm, rather than the mathematical algorithm itself. For example, a recent signature forging vulnerability CVE-2022-21449 in Java allowed an attacker to forge signatures for arbitrary messages because the implementation allowed the value (0,0) for a signature which should have been rejected as not being a member of the signature type [1].

Thus, proved-correct implementations are the first step towards proved-secure implementations of cryptographic primitives and algorithms. We attempt to reduce the vulnerabilities introduced by insufficient implementations of symmetric encryption schemes by generating formally proved-correct code for AES block cipher in its ECB and CBC modes of operations. We prove the correctness programatically using the Coq Proof Assistant.

III. Background

AES-128 encryption and decryption each consist of 10 rounds of various transformations which all operate on a state matrix. This state matrix is 4x4 bytes (128 bits) long and is initialized with the plaintext.

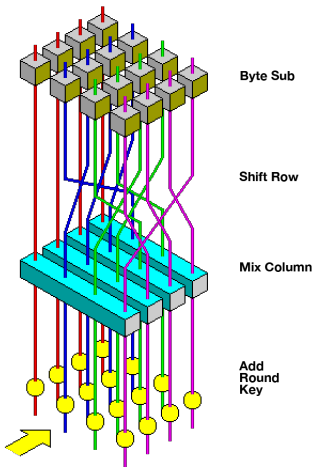

Fig. 1: *Typical AES encryption round*

Algorithm 1: AES-128 Encrypt

```
function AES-128-Encrypt (cipherkey, plaintext);
  Output: ciphertext
state = plaintext;
roundkeys = KeyExpansion(cipherkey);
state = AddRoundKey(state, roundKeys[0]);
for i ∈ {1, ..., 9} do
    state = SubBytes(state);
    state = ShiftRows(state);
    state = MixColumns(state);
    state = AddRoundKey(state, roundKeys[i]);
end
state = SubBytes(state);
state = ShiftRows(state);
state = AddRoundKey(state, roundKeys[10]);
return state
```

Algorithm 2: AES-128 Decrypt

```
function AES-128-Decrypt (cipherkey, ciphertext);
  Output: plaintext
state = ciphertext;
roundkeys = KeyExpansion(cipherkey);
state = AddRoundKey(state, roundKeys[10]);
for i ∈ {1, ..., 9} do
    state = InvShiftRows(state);
    state = InvSubBytes(state);
    state = AddRoundKey(state, roundKeys[10-i]);
    state = InvMixColumns(state);
end
state = InvShiftRows(state);
state = InvSubBytes(state);
state = AddRoundKey(state, roundKeys[0]);
return state
```

A. Security

Confusion is the idea that the relationship between the ciphertext (the encrypted output) and the key should be hidden. The *AddRoundKey* and *SubBytes* operations add confusion to AES.

Diffusion is the concept where modifying one bit of the plaintext, should affect around half of the bits in the ciphertext. The aim of *diffusion* is to hide the relationship between the plaintext and the ciphertext. The *ShiftRows* and *MixColumns* operations add diffusion to AES.

B. KeyExpansion - Round Key Generation

AES applies 10 rounds of encryption to the plaintext using different keys for each round. The round keys are computed as permutations of the main key using an algorithm called the *Key* Schedule. The key schedule actually generates 11 round keys, but the first round key is just the key itself that's XOR'ed with the plaintext before beginning the rounds of encryption. Since the key is 128 bits, it can be represented

as 4 words, of 4 bytes each. The Key Schedule includes three functions called *RotWord*, *SubWord* and *Rcon* that are applied to the fourth word of the key. *RotWord* simply shifts the word once to the left relative to the bytes as such: $(a_1, a_2, a_3, a_4) \rightarrow (a_2, a_3, a_4, a_1)$

SubWord takes a word as input, and applies the S-Box substitution to every byte in the word, and returns their result. *Rcon* is a function that XORs a word from the key with a preset round word, that's unique for every round. The round word essentially has the first byte being a preset round byte called the *Round Constant (rc)*, and the rest of the three bytes zeros, which looks like the following: $(rc, 0, 0, 0)$

For each round key generation, the functions *RotWord*, *SubWord* and *Rcon* are applied to the last word of the previous round key, and is XOR'ed with the first word of the previous round key to create the first word in the current round key. The second word is generated by xoring the second word of the previous round key with the first word of the current round key. The third and fourth words of the current round key are generated similarly to the second word. Figure 2 describes the key generation for the round keys.

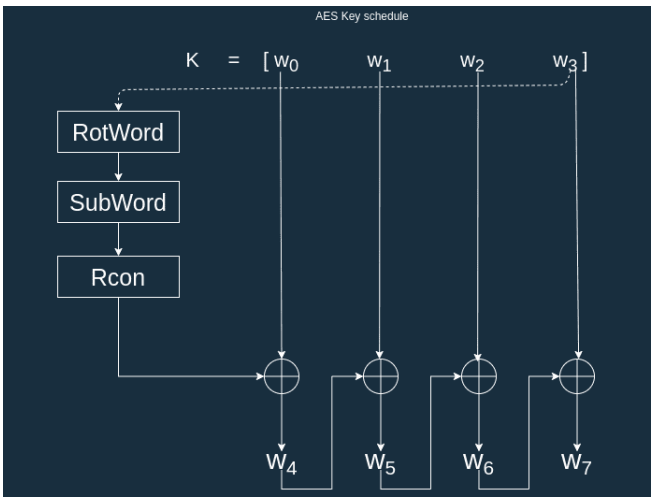

Fig. 2: Diagram describing the Key Schedule Algorithm

C. AddRoundKey

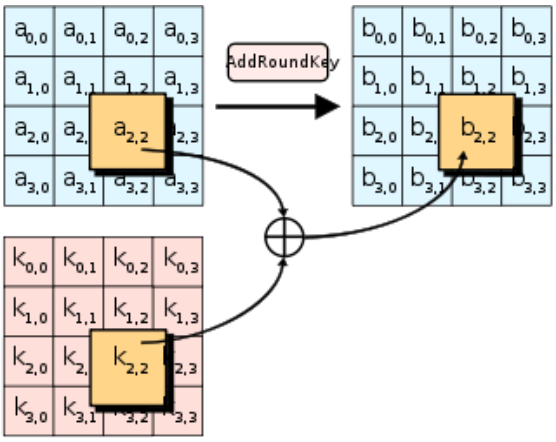

Fig. 3: *AddRoundKey*

AddRoundKey uses the generated round key (which has the same size as the state) and does an element wise xor with the state. It is important to note that the same function is used in the encryption pipeline and the decryption pipeline. The later operations we look at do not have this property, and need to have their own defined inverse function.

D. Galois Field

AES operates on numbers that exist in a finite field, also known as a Galois Field (GF). These field use the notation GF(x), where x is the size of the field which is a prime number or a prime power (p^y where p is prime). AES uses GF(256)/GF(2^8) for *MixColumns* and the S-Box in *SubBytes*.

Every element in the field is a polynomial represented by a number. For example the polynomial $x^4 + x^3 + x + 1$ would be represented by 0b00011011 or 27. Addition in GF(256) is the XOR operation on the byte representations and Multiplication is the product of the two polynomials modulo the characteristic modulus. AES uses 0x11B as its characteristic modulus, which can be shortened to 0x1B if the product is 8 bytes of less.

Galois Fields also have several mathematical properties that are relevant. Two we utilize in our implementation are:

1) Addition and multiplication are commutative and associative.
   $x + y = y + x \quad x * y = y * x$
   $x + (y + z) = (x + y) + z \quad x * (y * z) = (x * y) * z$
2) Multiplication is distributive over addition.
   $x * (y + z) = x * y + x * z$

E. SubBytes

SubBytes is a simple substitution operation that substitutes each byte in the state with a value in the S-Box, which acts like a lookup table. The S-Box is generated as the following:

1) Compute the multiplicative inverse (b) of the input byte c in GF(256).

2) Using $<<<$ as a left bitwise circular shift, we can calculate our substitution value s by $s = b \oplus (b <<< 1) \oplus (b <<< 2) \oplus (b <<< 3) \oplus (b <<< 4) \oplus 0x63$.

InvSubBytes reverses *SubBytes* by substituting each byte into the Inverse S-Box. We can generate the Inverse S-Box as follows:

1) Using the input byte s, $b = (s <<< 1) \oplus (s <<< 3) \oplus (b <<< 6) \oplus 0x5$.
2) Compute the multiplicative inverse of b.

In practice, it is more useful to use pre-generated tables instead of running this calculation for every input byte (or even generating it ourselves). These tables are each 256 bytes long, which is quite small.

F. ShiftRows

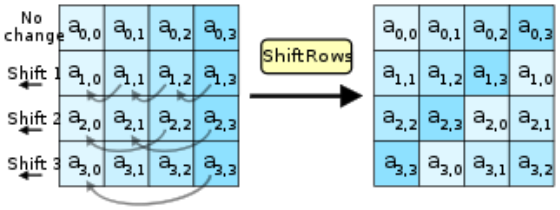

Fig. 4: *ShiftRows*

ShiftRows consists of circular left shifting the rows in the state matrix by its index (0-indexed). For example, Row 0 is not shifted and Row 2 is left shifted by 2. By shifting the rows, the columns change, and following *MixColumns* any change to the plaintext is diffused through the ciphertext.

The *InverseShiftRows* operation is exactly the same as *ShiftRows*, except every shift is a circular right shift.

G. MixColumns

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} \qquad 0 \leq j \leq 3$$

Fig. 5: *MixColumns*

MixColumns consists of multiplying the state matrix by an invertible matrix seen in Figure 5. Every operation is in GF(256), so every add is a XOR and elementwise multiply is closed by taking the modulus of the product. AES specifically uses $x^8 + x^4 + x^3 + x + 1$ or 0x11b as its modulus (meaning they divide the product by 0x11b and output the remainder).

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

Fig. 6: *InvMixColumns*

InvMixColumns is the same as *MixColumns* but multiplies by the inverse of the MixColumns matrix. Intuitively, multiplying a matrix by its inverse gives the identity matrix, which shows these functions are invertible. This matrix can be seen in Figure 6.

AES-128 block cipher can be used in a variety of modes-of-operations (ECB, CBC, CTR, GCM) in order to encrypt arbitrary size messages: . The ECB and CBC mode encryption and decryption algorithms are listed below as Algorithms 3,4,5 and 6.

Algorithm 3: AES-128-ECB Encryption

```
function AES-128-ECB-Encrypt (key, plaintext);
  Output: ciphertext
ciphertext = [];
for block in plaintext do
    ciphertext.append(AES-128-Encrypt(key,plaintext));
end
return ciphertext;
```

Algorithm 4: AES-128-ECB Decryption

```
function AES-128-ECB-Decrypt (key, ciphertext);
  Output: plaintext
plaintext = [];
for block in ciphertext do
    plaintext.append(AES-128-Decrypt(key,ciphertext));
end
return plaintext;
```

Algorithm 5: AES-128-CBC Encryption

```
function AES-128-CBC-Encrypt (key, iv, plaintext);
  Output: ciphertext
ciphertext = [iv];
for block in plaintext do
    blockcipher_input = block XOR iv;
    blockcipher_output =
      AES-128-Encrypt(key,blockcipher_input);
    iv = blockcipher_output;
    ciphertext.append(blockcipher_output);
end
return ciphertext;
```

Algorithm 6: AES-128-CBC Decrypt

```
function AES-128-CBC-Decrypt (key, iv, ciphertext);
  Output: plaintext
plaintext = [];
for block in ciphertext do
    blockcipher_output = AES-128-Decrypt(key,block);
    plaintext.append(blockcipher_output XOR iv);
    iv = block;
end
return plaintext;
```

IV. Implementing AES in Coq

A. AES-128 blockcipher

The first step to proving AES correctness was to implement AES-128 blockcipher. We implement the deterministic encryption and decryption functions by implementing types and functions for each step in the algorithm described in the previous section.

1) Types: To implement AES-128 blockcipher, we first defined several finite types that worked as the foundation for the data that the algorithm operated on. The types are: *bit*, *nibble*, *byte*, *word* and *matrix*. All of the types were created as finite Inductive types in Coq. Naturally, *bit* is a type with two values 0 and 1. *Nibble* is a type with 4 *bits*, and *bytes* is a type with 8 *bits*, and *word* is a type with 4 *bytes*. We represent the state of the data that goes through encryption and decryption, as well as the key, to be a *matrix* of 16 *bytes*.

```
Inductive bit : Set := s0 : bit | s1 : bit.
Inductive nibble : Set := bits4 : bit → bit
                  → bit → bit → nibble.
Inductive byte : Set :=
   bits8 : bit → bit → bit →
           bit → bit → bit →
           bit → bit → byte.
Inductive word : Set := bytes4 : byte →
           byte → byte → byte → word.
Inductive qword : Set := words4 : word →
           word → word → word → qword.

Inductive matrix : Set :=
   bytes16 : byte → byte → byte → byte →
             byte → byte → byte → byte →
             byte → byte → byte → byte →
             byte → byte → byte → byte →
             matrix.
```

2) Functions: We implement the AES-128 encryption and decryption functions by implementing rounds for each which use transformation functions that change the state matrix.

```
sub_bytes : matrix → matrix
inv_sub_bytes : matrix → matrix
shift_rows : matrix → matrix
inv_shift_rows : matrix → matrix
mix_columns : matrix → matrix
inv_mix_columns : matrix → matrix
add_round_key : matrix → matrix → matrix

enc_roundN: matrix → matrix → matrix
dec_roundN: matrix → matrix → matrix

enc_aes : matrix → matrix → matrix
dec_aes : matrix → matrix → matrix
```

For implementing sub_bytes and inv_sub_bytes we auto-generated Coq code using a python script that implemented a $s\_box : byte \rightarrow byte$ and a $inv\_s\_box : byte \rightarrow byte$ lookup table through which each byte of the state matrix gets passed. $shift\_rows$ and inv_shift_rows just rotates each row of the state matrix. $mix\_columns$ and $inv\_mix\_columns$ was written by implementing GF(256) multiplications with 01,02,03 for $mix\_columns$ and with 09,0b,0d,0e for $inv\_mix\_columns$. Multiplication with the larger factors was implemented using multiplication with the smaller factor 02. add_round_key is a simple XOR between the corresponding bytes of the key and the state matrices.

Each of the rounds of encryption and decryption use a different round key that is generated from the input key. This key generation happens through a key schedule which is implemented using sub functions rot_word, rot_word_inv, sub_word, sub_word_inv, $rcon\_word$ and $rcon\_word\_inv$ and each subsequent key depends on the previous round key.

Encryption and decryption rounds were implemented by stringing together the transformation functions above. AES-128 encryption and decryption itself was implemented by stringing together the rounds.

B. AES-128-ECB and AES-128-CBC

1) Types: For encrypting and decrypting arbitrary length messages, we implement a recursive type $blocks$ with two constructors: one corresponding to a single matrix block, another corresponding to a matrix block followed by further blocks.

```
Inductive blocks : Set :=
                B0 : matrix → blocks |
                BS : matrix → blocks →
                blocks.
```

2) Functions: We define corresponding ECB and CBC encryption and decryption functions which take a 128-bit key matrix as input. CBC mode is randomized and therefore also takes another 128-bit initialization vector matrix as input. Both ECB and CBC modes accept an arbitrary number of blocks as input and output the same number of blocks after performing the encryption or decryption using the AES-128 block cipher in the respective modes of operations.

```
enc_aes_ecb : matrix → blocks → blocks
dec_aes_ecb : matrix → blocks → blocks

enc_aes_cbc : matrix → matrix → blocks
          → blocks
dec_aes_cbc : matrix → matrix → blocks
          → blocks
```

V. Proving AES Correctness in Coq

A. Proving AES-128 Correct

1) Sub-proofs for state transformations: We prove the inverse properties for each of the transformations implemented on the state matrix:

```
sbytes_inv_sbytes : forall state : matrix,
   inv_sub_bytes (sub_bytes state) = state
srows_inv_srows : forall state : matrix,
   inv_shift_rows (shift_rows state) = state
mc_inv_mc : forall state : matrix,
   inv_mix_columns (mix_columns state) = state
xor_xor_matrix : forall state state' : matrix,
```

```
    add_round_key state' (add_round_key
       state' state)
    = state
```

Proof for $sbytes\_inv\_sbytes$ was done by proving that the s_box and inv_s_box were inverses for each of the 256 bytes and using it to rewrite the original goal. Proof for $srows\_inv\_srows$ was similarly straightforward by simple $destruction$ and $reflexivity$ tactics. Proof for xor_xor_matrix was done through proving the same property for more primitive types: bits and bytes and rewriting the original goal through those theorems. Proof for $mix\_columns$ was the most complicated and is explained in the section below.

2) MixColumns Sub-Proof: When proving MixColumns there were some fundamental theorems we needed to prove based on the approaches we considered.

We tried several approaches to defining the terms we needed. The difficulties primarily came from defining operations in GF(256). We were unable to find a suitable library that has properties of the field proven, so we had to prove, or try to prove, much of it ourselves. Addition was trivial but the multiplication was quite challenging. The two approaches we tried for multiplication were as follows:

1) Lookup Table: We initially used lookup tables, similar to our S-Box setup, that let us easily perform multiplication. Since we only needed to multiply an 8 bit input (256 elements) with 7 different numbers (2,3,4,9,11,13,14). This was a workable solution and we were able to easily prove that it was functional for some numbers, but an issue occurred when proving that *InvMixColumns* inverted *MixColumns*. Because it used a 128 bit input, we would need to brute force our proof with 2^{128} possible numbers. We actually had code to auto generate this proof, but it was computationally infeasible for us to run, and made the file huge. We ended up shifting to a new approach that led to our solution.
2) Defining multiplication for the 7 numbers we care about using double: We can easily define double as a shift left. Once this is done, if the high bit used to be 1 we mod by 0x1b. If the high bit used to be 0 we are finished. After this we were able to define all the multiplication operations we cared about by using the distributive property of multiplication over addition (XOR in GF(256)). Ex. $13 * b = (1 + 4 + 8) * b = (1 + 2^2 + 2^3) * b$. So using our double function this is equivalent to $b \oplus double(double(b)) \oplus double(double(double(b)))$.

Our 2 main approaches for proving that *InverseMixColumns* undoes *MixColumns* (*InvMixColumns*(*MixColumns*$(s)) = s$) were:

1) Brute Forcing the proof. This was something we considered, and what was necessary, if we wanted to use the precomputed table. We had code that would autogenerate the proof, but it was computationally infeasible to simulate a 2^{128} input.
2) There were several ways we could go about a proper proof, but they all eventually converge on the understanding that the multiplication of the *InverseMixColumns* matrix and the *MixColumns* matrix result in the identity matrix, which gives us our output. In order to properly prove this, we had a couple subproofs and theorems we needed to prove.

   a) The associative and commutative property of XOR. This was fairly simple and self explanatory, but very important since we needed this when proving the distributive property.
   b) Some other helper definitions we used include flattening a xor expression of the form $(a \oplus b) \oplus (c \oplus d) = ((a \oplus b) \oplus c) \oplus d$. This was extremely useful in simplifying expressions when proving the distributive property, as well as to set up for the next helper function. We had another theorem defined that combined the associative and commutative property such that $(a \oplus b)c = (a \oplus c)b$. Using this we can arbitrarily move terms left to right in a flattened expression.
   c) We proved that multiplying by 2 is distributive over addition by doing a case analysis on the high bit of the input. If the high bit is 0, we know that no modulus occurs and we can use the properties of xor to move terms around as the double will be a true double. If the high bit is 1, then the proof is a little more involved. Specifically we use the property of xor where $a \oplus a = 0$ to finalize the proof.
   d) We then proved the distributive property of multiplication over addition for numbers other than 2. Our definition of multiplication relies on this property, and by using this we can move terms around to make their cancellation easier. We used the defined distributive property of multiplication for multiplying by 2 and flattened the expressions. Once this happened we could easily move terms around with our xor commute and associativity functions. We initially proved this over 2 terms such that for $c \in \{1, 2, 3, 9, 11, 13, 14\}$, $c*(a \oplus b) = c * a \oplus c * b$ and used the two term distributive property to prove it for 4 terms. We needed 4 terms specifically because the state is a 4x4 matrix and multiplying the two matrices together leads to having distributive multiplication over 4 terms.
   e) Some interesting things we learned when doing this proof include how to strike a good balance between highly nested definitions and less nested definitions as well as shifting our multiplication from right associativity to left associativity. After these changes and rewriting some definitions, we were able to prove that *InvMixColumns*(*MixColumns*$(s)) = s$ without bruteforcing.

3) AES-128 Proof:

```
aes_correctness : forall k m : matrix,
    dec_aes k (enc_aes k m) = m
```

The proof for AES-128 correctness involved $unfold$-ing each of the encryption and decryption rounds and rewriting with the sub-proofs described above carefully and in-order. Each of the invertible transformations were beautifully cancelled out by their inverses by rewriting using sub-proofs.

B. Proving AES-128-ECB and AES-128-CBC Correct

The final goal was to prove correctness for arbitrary length messages.

```
aes_ecb_correctness :
forall (key : matrix) (message : blocks),
dec_aes_ecb key
    (enc_aes_ecb key message)
        = message

aes_cbc_correctness :
forall (key iv : matrix) (message : blocks),
dec_aes_cbc key iv
    (enc_aes_cbc key iv message)
        = message
```

This was done by using the $induction$ tactic in each case over the input message blocks followed by rewriting with $aes\_correctness$. For proving $aes\_cbc\_correctness$, the $generalize\ dependent$ tactic along with a xor-related $rewrite$ for the initialization-vector was also needed to move forward with the proof for the inductive case.

VI. Results

We were able to prove correctness for the AES-128 blockcipher as well as the AES-128-CBC and AES-128-ECB symmetric encryption algorithms. We started absolutely from scratch with the first Coq statement implementing the bit type, and eventually were able to prove the correctness property for a NIST standardized symmetric encryption scheme with sufficiently complex transformations (which introduce confusion and diffusion) for arbitrary length messages.

We extracted usable encryption and decryption code for OCaml, Haskell and Scheme languages for AES-128-CBC (an IND-CPA secure scheme) which we proved-correct in Coq.

VII. Conclusion

This formal proof of correctness for AES shows that if implemented correctly, AES decryption of a ciphertext deterministically returns the plaintext that had been encrypted under the same key.

VIII. Future Work

1) Asymmetric encryption: proving correctness for asymmetric schemes such as RSA, ElGamal, Cramer-Shoup using number theory and group theory axioms in Coq, and extracting usable code for the same.
2) Proving security: Proved-correct code is only one step towards vulnerability-free implementations. A much ambitious yet crucial goal is to prove security of cryptographic primitives under formal definitions such as IND-CPA, IND-CCA and more. Mathematically, this has been done for some schemes by reduction to a simpler but assumed secure property. For example, AES-128-CBC has been mathematically proven IND-CPA under the assumption that AES is PRF secure. IND-CPA and PRF are formal security definitions. Modelling these probabilistic definitions and generating proved-secure encryption and decryption code could eliminate a lot of cryptographic implementation vulnerabilities.
3) Extracting code for more languages: In fact, 2) above is an active research area and Project Everest [2] has been working on proven-secure cryptographic primitives in the TLS stack as well as extracting code for C and assembly languages. They work with the F* proof assistant instead of Coq which can extract C and assembly.

Acknowledgment

Thanks to Dr. Vivek Sarkar and the teaching staff for offering this inspiring course on Programming Languages.

References

[1] Madden, Neil. CVE-2022-21449: Psychic Signatures in Java. April 19, 2022. Available: https://neilmadden.blog/2022/04/19/psychic-signatures-in-java/

[2] Project Everest: https://project-everest.github.io/

[3] Duan, J., Hurd, J., Li, G., Owens, S., Slind, K., Zhang, J. (2005). Functional Correctness Proofs of Encryption Algorithms. In: Sutcliffe, G., Voronkov, A. (eds) Logic for Programming, Artificial Intelligence, and Reasoning. LPAR 2005. Lecture Notes in Computer Science(), vol 3835. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11591191_36