CRYPTOGRAPHIC BLOCK CIPHERS IN FUNCTIONAL PROGRAMMING: A CASE STUDY ON FELDSPAR AND AES

Gregor Ulm

May 31, 2016

CONTENTS

ABSTRACT

Cryptographic block ciphers are defined as mathematical functions, and thus a prime candidate for implementation in functional programming languages. We implement the block cipher *Rijndael*, which was selected as the block cipher used for the Advanced Encryption Standard (AES), in Cryptol, Haskell, Feldspar, and C. We analyze how well Feldspar, a language originally designed for digital signal processing, is suited to this task. We highlight relative strengths and weaknesses of Feldspar for implementing cryptographic block ciphers, and suggest possible improvements to this language.

1 INTRODUCTION

Block ciphers are common in cryptography. Furthermore, they are defined as mathematical functions, which gives rise to the temptation of implementing them in functional programming languages. Thus, in our exploratory study we implement the main part of the block cipher Rijndael, which was adopted as the Advanced Encryption Standard (AES) by the US government. While AES is no longer a cutting-edge block cipher, it is still popular and has been studied extensively.

The main goal of our work is to analyze the potential of the embedded domain-specific language (DSL) Feldspar for implementing cryptographic block ciphers. Our analysis takes place on several levels, and considers four different programming languages, namely:

- Cryptol, a DSL designed for analyzing and implementing cryptographic algorithms
- Haskell, a general-purpose functional programming language
- Feldspar, a DSL embedded in Haskell, originally designed for digital signal processing and parallelism
- C, the stalwart systems programming language

We are going to perform comparisons on the following levels:

- Cryptol source code vs. Feldspar source code
- Haskell source code vs. Feldspar source code
- Handwritten C source code vs. C source code generated by Feldspar

Our analysis has the overall goal of exploring the suitability of Feldspar for implementing cryptographic block ciphers. After assessing relative strengths and weaknesses, we will make suggestions for future work, some of which is possibly suitable for future student research projects.

In summary, our contribution is as follows:

- Perform a comparative study on a syntactic level with regards to implementing cryptographic block ciphers
- Present a non-trivial Feldspar program
- Provide an analysis of strengths and weaknesses of Feldspar for the chosen domain
- Identify potential improvements for Feldspar

The structure of this paper is as follows: We assume that the reader is already familiar with Haskell and C. Thus, we will briefly introduce only Cryptol and Feldspar in section 2 and 3, respectively. This is followed a whirlwind tour through relevant and necessary cryptographic background knowledge in section 4. Section 5 presents a high-level description of AES, but omits implementation details. Those are illustrated in section 6, which shows parts of AES implemented in Cryptol, Haskell, Feldspar, and C. In section 7 we discuss our experience with Feldspar in the context of our work, highlighting perceived advantages and disadvantages. This is followed by recommendations for future work in section 8. The appendix contains source code of AES implementations in Cryptol, Haskell, Feldspar, and C.

2 CRYPTOL

Cryptol is a domain specific language for specifying, implementing, and verifying cryptographic algorithms, which has been developed by Galois, Inc.[1] The first version of Cryptol was released in 2003, and was proprietary [17, 18]. Only a rather restricted trial version had been made available to the public. On the other hand, the successor Cryptol 2, which has been developed from scratch, has been open-sourced in 2013. Cryptol is written in Haskell, but it is not embedded in that programming language. As a consequence, it is not possible to directly call Cryptol in a Haskell program. Relevant for this paper is that the first version of Cryptol was able to generate C source code. Cryptol 2 currently does not have this feature, but a code generator may eventually be added.[2]

From a programming-language theoretic perspective it is noteworthy that the type system of Cryptol is advanced, as it implements sized types on top of Hindley-Milner type inference. Sized types are an instance of dependent types. Dependent types can depend on arbitrary values, while sized types merely encode the size of a data type. For instance, take a function for reversing a string. A string is a list of characters, like in Haskell. Unlike Haskell, though, Cryptol can encode a length argument, so that the type checker is able to verify that the resulting string has the same length as the input string. If the type checker is not able to verify this property, such a function is rejected as erroneous.

The previously mentioned property can be encoded in a type signature like `reverse :: String n -> String n`, where n stands for the length of the input string as well as the length of the output string, which have to be identical. Thus, the type checker will attempt to verify that the length of the string that results from applying `reverse` has the same length as the input string. By comparison, the type signature of a corresponding Haskell program is `reverse :: String -> String`. A Haskell programmer wanting to verify that this function returns a string of the same length as the input string would either need to resort to experimental advanced type system features or add an assertion that is checked at runtime.

The syntax of Cryptol is very similar to Haskell. However, since Cryptol is not a general-purpose programming language, the number of primitives and build-in functions is small, as they were designed with cryptography in mind. Concretely, support is provided for arithmetic, comparisons, bit operations, and various standard operations on sequences. Unusual are built-in operators for operations on polynomials for multiplication, division, and taking the modulus, which are relevant for cryptography. Therefore, it is possible to very concisely express cryptographic algorithms in Cryptol.

As restricted as the provided operators and in-built functions are the provided data types. Cryptol knows only four basic data types: bits, arbitrarily deeply nested sequences (linked lists), tuples, and records. Noteworthy syntactic features are regular and parallel sequence comprehensions as well as support for infinite sequences. Similar to Haskell, a string in Cryptol is a sequence of characters.

Cryptol has drop-in support for SAT and SMT solvers, such as Microsoft's Z3 [10], which makes it possible to find rigorous formal proofs of properties

1 The website of Galois is located at `http://galois.com`.
2 This statement is based on response to a question we asked on the Cryptol users mailing list, which was answered by Galois employee Aaron Tomb (January 26, 2016).

that are satisfied by source code. Furthermore, randomized property testing is supported, similar to the Haskell library QuickCheck [5].

3 FELDSPAR

Feldspar [2] is a recent embedded domain specific language, which is hosted in Haskell. It was originally a joint project between Ericsson AB, Chalmers University of Technology in Gothenburg, Sweden, and Eötvös Loránd University in Budapest, Hungary. Currently, it is mainly developed at Chalmers, with contributions from the Swedish Institute of Computer Science and Ericsson. It is noteworthy that Ericsson has explored the suitability of Feldspar for baseband channel estimation in the context of a research project.[3]

Feldspar has been designed for implementing algorithms in the digital signal processing (DSP) domain as well as parallel programming — note the letters *PAR* in the name of this programming language. Relieving programmers of the burden of having to write C code manually is one of the declared goals of Feldspar. Therefore, it aims to generate readable C code, as opposed to optimized C code. For instance, Feldspar does not unroll all loops, as is evinced by the presence of several for loops in our generated C code much later on. Sample programs show that C code generated with Feldspar compares favorably with some C reference implementations [11]. As we will later on show, we do not fare quite so well with the C code we generate from Feldspar.

Feldspar shares some of the advantages of Haskell, such as static typing and Hindley-Milner type inference, or higher-order functions. The language itself is purely functional. However, programming in it, coming from a Haskell background, is not necessarily straightforward. The general programming model is based on the dataflow model, which goes back to research by Dennis in the 1960s [8, 9]. One of the motivations behind research in dataflow programming was parallel programming. Likewise, the authors of Feldspar consider support for parallelism a major long term goal [2].

As a consequence of the dataflow programming approach, Feldspar poses some limitations which are not present in its host language Haskell. Concretely, explicit recursion is not directly supported, but would need to take place on the meta-level, in Haskell code. On the other hand, Feldspar provides operators for folding, as well as constructs for while and for loops. Furthermore, compared to general purpose programming languages, there is only limited support for control flow. Lastly, Feldspar operates on vectors, allowing the programmer to specify whether a vector is merely symbolic or supposed to exist in memory.

3 For context, refer to the presentation slides of David Engdal's guest lecture at Chalmers University of Technology on October 5, 2010, which are available at `http://www.cse.chalmers.se/edu/year/2010/course/TDA555_Introduktion_till_funktionell_programmering/Material/Guest1/david.pdf` (accessed May 31, 2016).

4 SOME THEORETICAL FOUNDATIONS

4.1 Symmetric Encryption in a nutshell

The motivation for cryptography is generally presented as the story of Alice and Bob who would like to share secret messages in the presence of formidable opponents such as the eavesdropper Eve. The question arises how Alice and Bob could share secret messages successfully. Following Shannon's seminal work [20], the problem of secure message sharing can be specified as a *secrecy system*, which has the following properties.

Alice holds a message M, which she enciphers, using a key k. The enciphered text is the result of a transformation, namely $E = T_i M$, where T_i is a transformation that uses key i. Bob has to be able to receive the key and the enciphered message separately. Of particular relevance is the *one-time-pad* encryption technique, which uses a random secret key of the same length as the plaintext M. Note that both M and k are represented as a sequence of bits, where the latter is a sequence of *random* bits. The cipher text results from *xor*ing the plaintext with the key. As a consequence, for Eve the resulting cipher text is indistinguishable from a random sequence of bits.

Shannon [20] proved the one-time pad to be "information-theoretically secure", since it does not convey any information at all about M. Even the length of the message can be concealed by padding, i.e. adding irrelevant bits to M. Assuming true randomness of the key, there is no plausible cryptographic attack for the one-time pad. Attempting to use brute force to guess the one-time pad would be dubious at best, since one could instead attempt to brute force the original message instead.

There are practical reasons why the one-time pad is less relevant. One reason is indeed that the key is as long as the original message. One may therefore safely assume that if it is possible for Alice to securely transmit the key to Bob, then she could as well have securely transmitted the original message instead. Generally, it is easier to securely transmit a short key than a potentially much longer message. Thus, Eve is assumed to be able to get hold of the encrypted message E, but not of key i. On the other hand, Bob is in possession of both the key and the encrypted message E. In order to decrypt E, he has to decrypt the message with the transformation $M = T_i^{-1} E$. This requires that any transformation T_i has a unique inverse T_i^{-1}, such that $T_i T_i^{-1} = I$, which is the identity transformation. This is the key feature of *symmetric* encryption.

Assume Eve was able to get the ciphertext, but not the corresponding key. Since she does not possess the key, she has to find a different way to decipher the message. The key we are using is intended to approximate information-theoretical properties of the one-time pad, which, as stated above, is a random sequence of bits. Thus, for Eve the ciphertext should look as if it was encoded with a one-time pad, even though in reality it was encoded with a key that merely produced a *seemingly* random sequence of bits — we will see examples in a later subsection on block ciphers. This seemingly random sequence of bits should ideally be indistinguishable from ciphertext that results from using the one-time pad. Bob can later on use the same key to decrypt the ciphertext in order to recreate the plaintext. For examples on how to use a key that is shorter than the plaintext, please refer to section 4.4, which presents several block cipher modes.

Thus, for practical reasons, the goal of cryptography is to use an encryption that will thwart Eve long enough so that by the time she has figured out the key by brute force, the original message she reconstructed will have long ceased to be of any importance. Note that Shannon [20] points out that Eve shall be assumed to have knowledge of the transformations T_i as well as the associated probabilities of the keys. Of course, brute force could be used to determine the exact key i. For this reason, the probability space has to be sufficiently large in order to make such an attempt infeasible.

4.2 Sum and product ciphers

In a discussion on the algebra of secrecy systems, Shannon [20], describes sum and product ciphers. Block ciphers are based on the idea of an *iterated* product cipher. However, before we get to that point, let us briefly retrace the theoretical development. The starting point for each of the subsequent methods is the insight that two secrecy systems T and R can be combined in different ways, with the goal of creating a new secrecy system S.

Considering probabilities p, q, where $p + q = 1$, and secrecy systems T, R, which have the same domain, a new secrecy system S could be created in, for instance, the following way: $S = pT + qR$. This is a sum cipher. Concretely, this means that the probabilities p, q are used to create a *new* key S that indicates which of R, T are used. The total key of S specifies which of T or R, and which key, is used. Of course, any number of probabilities and transformations can be used, as long as the sum of the probabilities is 1.

Product ciphers are more complex as they entail applying transformations in sequence. Thus, they are created by sequential composition of other ciphers. For instance, one could define an operation $S = RT$, indicating that first T is applied, and then R. The key of S consists of two keys, taken from R and T, where R_i has probability p_i, and similarly for S. In the next subsection we discuss block ciphers. Those are sequences of rounds and thus product ciphers.

4.3 Block ciphers

The following description of block ciphers follows Ferguson et al. [14]. Block ciphers encrypt fixed-size blocks of data, such that a block of the original message and a block of the encrypted message have the same size. Thus, a block cipher is a function that maps a key of m bits and a block of n bits to a new block of n bits. We will encounter a concrete example in the next section, which presents a high-level description of AES. For now, suffice it to say that block ciphers perform repeated transformations of a block of data.

Similar to the transformations we have mentioned in section 4.1, block ciphers are reversible, which means that for any block cipher T_i that encrypts a message, there is a corresponding transformation T_i^{-1} that, when used on the encrypted message, recreates the original unencrypted message. This is a high-level view, though. On a much lower level of detail, a block cipher maps a key and an block that are given as input to a new block as output.

As Ferguson et al. [14] remark, there is no commonly agreed upon definition of a block cipher, so they instead describe an *ideal block cipher*. Concretely, such an ideal block cipher does the equivalent of independently assigning a unique random permutation to each key value. If one had a 128-bit block cipher, i.e. one specific permutation on 128-bit values, this could be illustrated by having, for each key value, a lookup table of 2^{128} elements of 128 bits each. A block cipher would then select one of those tables uniformly at random. Of course, real block ciphers do not use lookup tables, as this would be infeasible.

The previous illustration is impractical for another reason [14]: once the tables are determined, the ideal cipher would no longer be random, and thus no longer be ideal. Since there are no ideal block ciphers, we focus on AES, a popular real block cipher. Like many block ciphers, AES is a product cipher that consists of a sequence of rounds. Each round may be regarded as a weak block cipher. This refers back to the previously mentioned, in our discussion of product ciphers, sequential composition of ciphers, i.e. several transformations of the input message are performed by applying a possibly weak block cipher. Several *rounds* of weak block ciphers are applied in order to create a strong block cipher.

We have seen that block ciphers are designed with the goal of approximating the one-time pad. Related to this goal are the concepts *confusion* and *diffusion* [20]. In an information-theoretical context, confusion signifies that there is a drastic change between the plaintext and the ciphertext. This is a consequence of each bit of the ciphertext depending on several bits of the key. Diffusion means that changing one bit of the input leads to the change of many bits in the output. Ideally, changing one bit of the plaintext should lead to the change of half of the bits of the entire ciphertext. Thus, a block cipher of significance for practical cryptography would need to exhibit both confusion and diffusion.

4.4 Block cipher modes

After describing symmetric encryption and block ciphers in some detail, we still need one more piece in order to encrypt a message. The issue is that block ciphers operate only on one block of data. However, any message we would want to encrypt is likely to be much larger than that. Thus, a *block cipher mode* has to be chosen in order to encrypt a message. Concretely, a block cipher mode is a function that encodes a plaintext message by applying a block cipher repeatedly in a particular way. There are several block cipher modes, but in the following we describe only three of the five block cipher modes recommended by the NIST for use with AES, following Ferguson et al. [14] as well as Dworkin [12].

The *counter* (CTR) mode is most relevant for our previous discussion of the one-time pad. It uses a *nonce*, i.e. an arbitrary number that may only be used once, as its initialization vector. Subsequent blocks of the key stream are encoded using the nonce and a counter, which can be any function. Of course, the counter function should not repeat itself for a long time, if at all. Thus, a common choice is a function that increments the counter by one for each new plaintext block. Using a secret key k, the cipher text is computed via $C_0, C_1, ..., C_{n-1} = enc_k(< 0 >) \oplus M_0, enc_k(< 1 >) \oplus M_1, ..., enc_k(< n - 1 >) \oplus M_n$, where $enc_k(B)$ is the block cipher encryption of block B using key k, $< m >$ is a block that contains the binary encoding of natural number m, $\oplus$ denotes bitwise *xor*, and M_i are the blocks of the plain text message. The central idea behind the CTR mode is to generate a stream of seemingly random bits $enc_k(< 0 >), enc_k(< 1 >), ..., enc_k(< n-1 >)$. For an adversary who is not in possession of k, this sequence is practically indistinguishable from a completely random sequence of bits, considering a non-repeating counter function. The cipher text is created by *xor*ing the plain text with the generated stream of bits, just like it is the case with one-time-pad encryption. Thus, the counter mode approximates the one-time pad.

A fundamental property of CTR is that being able to distinguish a seemingly random stream of generated bits from a stream of completely random bits would allow the adversary to break the block cipher. Similarly, being able to learn something about the plain text based on the cipher text would make it possible to break the underlying cipher. Using these properties, it can be shown that CTR is not weaker than the underlying block cipher. Lastly, note that CTR has the added benefit that both encryption and decryption are fully parallelizable, i.e. different blocks can be independently encrypted and decrypted.

After having covered CTR in some detail, we will briefly discuss two more of the block cipher modes NIST recommends for use with AES. Arguably the simplest of them is the *electronic codebook* (ECB), which encrypts each block of the plaintext separately. The downside is that identical blocks of the plain text lead to an identical block of cipher text, which leads to this block cipher mode being considered flawed. On the other hand, ECB can be fully parallelized for both encryption and decryption since each block is encrypted separately.

The *cipher feedback* (CFB) mode of operation, normally referred to as n-bit CFB, requires an initialization vector. CFB uses a bit shift register, which is initialized with an initialization vector. First, the shift register is encrypted with the block cipher. Afterwards, the highest n bits of the result are *xor*ed with n bits of plaintext, thus generating n bits of ciphertext. With a bitwise left shift, the resulting n bits are then shifted into the shift register. CFB encryption cannot be parallelized. On the other hand, decryption is parallelizable.

4.5 Testing block ciphers

Testing that an implementation of a block cipher conforms to its mathematical specification is difficult. While it is possible, for instance in Cryptol, to test whether certain properties of functions hold, either via property-based testing or theorem proving, an implementation of a block cipher is considered correct not via discovering mathematical proofs, but after passing test vectors. For instance, the NIST provides test vectors for AES, i.e. sample inputs and their corresponding outputs. Thus, testing an implementation via test vectors is an example of unit testing, which leads to Dijkstra's famous dictum, first quoted in [4], that testing only ever shows the presence of errors, but never their absence.

As a consequence, testing an AES implementation provides *some* certainty, but by no means complete certainty that the implementation is indeed correct. In practice, though, this distinction may be of little relevance. In fact, the NIST maintains a list of validated AES implementations,[4] which is based on two conditions: passing all test vectors as provided in the standard, which was described in [19], and doing so in an accredited laboratory for cryptographic and security testing.

4 NIST. "Advanced Encryption Standard Algorithm Validation." NIST.gov. `http://csrc.nist.gov/groups/STM/cavp/documents/aes/aesval.html` (accessed March 1, 2016).

Implementations of block ciphers are often tested via comparing the output of an implementation with the output that is generated by a so-called gold standard implementation. If the results of such an implementation match all results that are produced by a gold standard implementation, the former is assumed correct. It is also possible to formally verify a new implementation against an existing gold standard implementation.

5 HIGH-LEVEL DESCRIPTION OF AES

In the early 2000s, the US Government adopted the block cipher Rijndael, named after its authors Vincent Rijmen and Joan Daemen, as the Advanced Encryption Standard (AES) [19]. A detailed description of Rijndael has been published by Rijmen and Daemen themselves [7]. The description below, however, is mostly based the much more high-level presentation of Katz and Lindell [15], since our goal is not to give a complete specification that could be used as the basis of an implementation, but merely to convey the general idea behind AES.

The two stated goals of AES are invertibility and simplicity. The former is expressed by the fact that this block cipher is a symmetric-key algorithm, i.e. the same key encrypts and decrypts data. The latter is reflected by four relatively simple operations that operate on the *state*, which is a 4-by-4 column-major order matrix of bytes. Each such byte represents an element in a Galois field. AES furthermore exhibits some flexibility. While the block size is 128 bits, there are three possible key lengths: 128, 192, and 256 bits, with the larger key sizes being used for particular applications that require a larger key space.

AES consists of a number of rounds, i.e. iterative applications of block ciphers. The exact number of rounds depends on the length of the used key. 128, 192, and 256-bit keys entail $n = 10, 12$, and 14 rounds, respectively. Those rounds are divided into four distinct stages. First, in a *key expansion* stage a round key is derived from the cipher key. This is followed by an initial round, which calls `AddRoundKey`. The next $n-2$ rounds, normally referred to as AES *main rounds*, successively apply the following four functions to the current state:

- `SubBytes`: substitute bytes according to a lookup table S
- `ShiftRows`: shift row i of the state cyclically by i steps
- `MixColumns`: multiply each column by a fixed polynomial
- `AddRoundkey`: *xor* round key and state

In a final round, only `SubBytes`, `ShiftRows`, and `AddRoundKey` are applied to the state.

The substitution table that is used for `SubBytes` is called the *Rijndael S-box*. Its values are predefined in the AES standard. It is possible to use different S-boxes. In fact, because there are 256 different values, the total number of possible S-boxes for AES is the factorial of 256. However, it is generally not recommended to deviate from the standard, as alternative S-boxes may be less resilient to cryptographic attacks. In fact, Courtois et al. [6] introduce a measure of strength for S-boxes. Very weak S-boxes even lead to a linear transformation of their input. Note that the other transformations in AES are linear, so the Rijndael S-box is the only source of non-linearity, and thus of the aforementioned concept of confusion; diffusion is provided by the other functions. Tampering with the S-box may therefore lead to undesirable results from a cryptographic perspective.

6 IMPLEMENTING AES

This section presents implementations of the main round of AES-128 in Cryptol, Haskell, Feldspar, and C, both handwritten, and as it is generated by Feldspar. We will highlighting differences at the source code level and the generated C code, where applicable. The complete code in Cryptol, Haskell, Feldspar and C is provided in an appendix. C generated by Feldspar is not provided in full due to its verbosity, however.

6.1 Cryptol

The DSL Cryptol has been designed for implementing cryptographic algorithms, which leads to the resulting code being rather concise. This is due to two reasons: First, the influence from Haskell, and, second, the fact that many operators that are relevant for cryptography are built-in.

The code below, as well as the listing presented in the appendix, largely follows the AES implementation given in the official Cryptol tutorial [13]. The most significant deviation is our implementation of `MixColumns` as table lookups.

The state is modeled as a 4-by-4 matrix of bytes, representing elements of a 256-element Galois field. A key in AES-128 is defined identically.

```
type State = [4][4][8]
```

The function `AESMainRound` implements a single main round of AES, as described previously. An AES main round consists of the subsequent application of the functions `SubBytes`, `ShiftRows`, and `MixColumns` to a state, followed by `AddRoundKey`:

```
AESMainRound : (State, State) -> State
AESMainRound (rk, s) = AddRoundKey (rk, MixColumns (ShiftRows (SubBytes s)))
```

In detail, the functions used in `AESMainRound` are as follows. `SubBytes` is commonly implemented as a simple lookup function, where each element in the state is replaced by its corresponding value in a predefined S-box:

```
SubBytes : State -> State
SubBytes state = [ [ SubByte b | b <- row ] | row <- state ]
    where SubByte x = sbox @ x

sbox : [256][8]
sbox = [ 0x63, 0x7c, ..., 0x16 ]
```

The Cryptol code is straightforward: In the sequence comprehension, the rows are extracted from the state. From each resulting row, we extract all bytes. Those bytes are then substituted via lookup in an S-box, using the `subByte` function. The expression `sbox @ x` signifies a lookup operation of index `x` in the sequence `sbox`, which is given in incomplete form. Thus, each of the 16 bytes of the state is replaced by its corresponding entry in the provided S-box. Note that each byte `b` in the state is one of $2^8 = 256$ values, which is used to index into an S-box with 256 entries.

In `ShiftRows`, shown below, a parallel sequence comprehension is used, i.e. a comprehension with multiple arms, where the contents of each arm will be zipped. This code is taken from the official Cryptol tutorial [13].

```
ShiftRows : State -> State
ShiftRows state = [ row <<< shiftAmount
                  | row <- state
                  | shiftAmount <- [0 .. 3]
                  ]
```

The sequence comprehension in the code above has two arms, indicated by an expression to the right of a pipe symbols (|). Thus, the rows and shift amounts are zipped and then consumed by the expression `row <<< shiftAmount`. The operation for rotating bytes, <<<, is a primitive operation in Cryptol. The value `shiftAmount` specifies by how much a row of bytes has to be cyclically shifted to the left. The first row is shifted by 0, the second row by 1, the third row by 2 and, finally, the last row by 3.

Eventually, we chose a different implementation of `ShiftRows`, which encodes the previous steps more explicitly and arguably improves readability:

```
ShiftRows : State -> State
ShiftRows [ row1 ,
            row2 ,
            row3 ,
            row4 ] = [ row1 <<< 0 ,
                       row2 <<< 1 ,
                       row3 <<< 2 ,
                       row4 <<< 3 ]
```

We pattern match on each row, and specify the shift amount. Of course, the expression `row1 «< 0` simplifies to `row1`, since no values are shifted.

`MixColumns` consists of matrix multiplication in a finite field. Note that for performance reasons, this step is often implemented as a table lookup in AES, which was also our choice in the subsequent implementations in Haskell and Feldspar. The definition of `MixColumns` below captures that a fixed matrix m is multiplied with the state, where `gf28MatrixMult` is a user-defined function for performing matrix multiplication in a finite field. In the official Cryptol tutorial [13], the definition below is given:

```
MixColumns : State -> State
MixColumns state = gf28MatrixMult (m, state)
    where m = [[2, 3, 1, 1],
               [1, 2, 3, 1],
               [1, 1, 2, 3],
               [3, 1, 1, 2]]
```

However, our final implementation implements this step via table lookups:

```
MixColumns : State -> State
MixColumns [ [a1, a2, a3, a4] ,
             [b1, b2, b3, b4] ,
             [c1, c2, c3, c4] ,
             [d1, d2, d3, d4] ] = transpose [ MixCol [ a1, b1, c1, d1 ] ,
                                              MixCol [ a2, b2, c2, d2 ] ,
                                              MixCol [ a3, b3, c3, d3 ] ,
                                              MixCol [ a4, b4, c4, d4 ] ]

MixCol : [4][8] -> [4][8]
MixCol [a0, a1, a2, a3] = [b0, b1, b2, b3]
    where b0 = mult2 @ a0 ^ mult3 @ a1 ^         a2 ^         a3
          b1 =         a0 ^ mult2 @ a1 ^ mult3 @ a2 ^         a3
          b2 =         a0 ^         a1 ^ mult2 @ a2 ^ mult3 @ a3
          b3 = mult3 @ a0 ^         a1 ^         a2 ^ mult2 @ a3
```

```
mult2 : [256][8]
mult2 = [ 0x00, 0x02, ..., 0xe5 ]

mult3 : [256][8]
mult3 = [ 0x00, 0x03, ...  0x1a ]
```

In `MixColumns` we make use of pattern matching, which allows us to easily extract a column of values. We transpose the resulting sequence of sequences, in order to recreate the correct order of the values, due to `MixCol` returning column values as a one-dimensional sequence. In that function, the operator `@` is one we are already familiar with. It is used for looking up precomputed values for multiplications in the tables `mult2` and `mult3`, whose definitions are given in an abbreviated form above. The caret symbol (ˆ) performs bitwise *xor*ing.

Finally, `AddRoundKey` is simply an *xor* operation of the round key and the state:

```
AddRoundKey : (State, State) -> State
AddRoundKey (rk, s) = rk ^ s
```

The conciseness of `AddRoundKey` is due to operator overloading of the caret symbol (ˆ), which is now applied to an entire sequence. Note that the type system of Cryptol checks that both sequences `rk` and `s` have the same length and shape. In Haskell and Feldspar, `AddRoundKey` requires the function call `zipWith xor`.

6.2 Haskell

In Haskell, we chose to model the state as a list of lists containing type `Word8`:

```
type State  = [[Word8]]
```

Due to point-free style, `aesMainRound` is arguably more elegant than its equivalent definition in Cryptol, which does not support this kind of function composition:

```
aesMainRound :: State -> State -> State
aesMainRound rk = addRoundKey rk . mixColumns . shiftRows . subBytes
```

As in the previous implementation in Cryptol, we use a predefined S-box and perform a simple lookup in the `SubByte` operation, which is shown below.

```
subBytes :: State -> State
subBytes = map row
    where row = map subByte

subByte :: Word8 -> Word8
subByte x = sbox ! x

sbox :: Array Word8 Word8
sbox = array (0, 255) $ zip [0..] vals
    where vals = [ 0x63, 0x7c, ..., 0x16 ]
```

In the code we have just seen, `sbox` is a Haskell `Array`. This choice was made for performance reasons, since Haskell arrays offer constant time lookup, while a list would require linear time lookup. Lists are used otherwise in order to enjoy the convenience that is provided by pattern matching and list comprehensions. Note that the explanation mark operator (`!`) stands for the indexing operation on arrays, i.e. the expression `sbox ! x` states that we extract the element at index `x`. In the example below, we chose to 0-index the array. Due to the fact that the type `Word8` can be inhabited by 256 values, 0 and 255 are thus the bounds of the indices of the array `sbox`.

The `shiftRows` operation can be implemented in various ways. One solution, making use of Haskell list comprehensions, is as follows:

```
shiftRows :: State -> State
shiftRows xss = [ shift n xs | (n, xs) <- zip [0..] xss ]
     where shift n xs = drop n xs ++ take n xs
```

The approach of `shiftRows` above is quite similar to the first of the two implementation of this function in Cryptol that we have seen before. We extract each row from the state (`xss`), zip it with a shift amount ranging from 0 to 3, inclusive, and then call the helper function `shift`, which performs a cyclic rotation of the row, depending on the shift amount `n`.

The following alternative definition of `shiftRows` uses pattern matching, which arguably leads to more readable code.

```
shiftRows :: State -> State
shiftRows [[a1, a2, a3, a4],
           [b1, b2, b3, b4],
           [c1, c2, c3, c4],
           [d1, d2, d3, d4]] = [[a1, a2, a3, a4],
                                [b2, b3, b4, b1],
                                [c3, c4, c1, c2],
                                [d4, d1, d2, d3]]
```

In Haskell, `mixColumns` via table lookups is a rather straightforward operation:

```
mixColumns :: State -> State
mixColumns = transpose . map mixColumn . transpose

mixColumn :: [Word8] -> [Word8]
mixColumn [a0, a1, a2, a3] = [b0, b1, b2, b3]
    where b0 = mult2 ! a0 `xor` mult3 ! a1 `xor`         a2 `xor`         a3
          b1 =         a0 `xor` mult2 ! a1 `xor` mult3 ! a2 `xor`         a3
          b2 =         a0 `xor`         a1 `xor` mult2 ! a2 `xor` mult3 ! a3
          b3 = mult3 ! a0 `xor`         a1 `xor`         a2 `xor` mult2 ! a3


mult2, mult3 :: Array Word8 Word8
mult2 = array (0, 255) $ zip [0..] vals
    where vals = [ 0x00, 0x02, ..., 0xe5 ]

mult3 = array (0, 255) $ zip [0..] vals
    where vals = [ 0x00, 0x03, ..., 0x1a ]
```

This definition of `mixColumns` illustrates one advantage of using a list of lists for modeling state, i.e. we can easily get access to the columns via transposing. As we will soon see, Feldspar is not as convenient in that regard. The arrays `mult2` and `mult3` contain hardcoded values for multiplication with 2 and 3. Thus, `mixColumn` performs matrix multiplication via table lookups. In the where-clauses, *xor* operators are used for adding bytes.

The function `AddRoundKey`, which was short and elegant in Cryptol, is slightly more cumbersome to encode in Haskell, requiring a nested `zipWith`:

```
addRoundKey :: State -> State -> State
addRoundKey  = zipWith (zipWith (xor))
```

6.3 Feldspar

In Feldspar, we model state as a two-dimensional pull vector,[5] containing values of type `Word8`:

```
type State =  Pull DIM2 (Data Word8)
```

The type of `State` expresses that it is a two-dimensional pull vector of type `Data Word8`. `Data a` expresses a Feldspar type, as opposed to a Haskell type, where `a` is the value computed by the program [2]. In this example, `a` is instantiated as `Word8`, which is an 8-bit unsigned integer type representing a byte.

Pull vectors are arrays defined as functions that map an index to its corresponding element. In the generated code, pull vectors may be eliminated due to *fusion*, which is the removal of immediate vectors. An example of a concrete definition of a pull vector is given further below, in the function definition of `sbox'`.

The definition of `aesMainRound` is superficially identical to the one given in Haskell in the previous subsection. However, keep in mind that the type of `State` in Haskell and Feldspar is not identical.

```
aesMainRound :: State -> State -> State
aesMainRound k = addRoundKey k . mixColumns . shiftRows . subBytes
```

The `subBytes` operation below can simply be mapped over the entire pull array:

```
subBytes :: State -> State
subBytes = map subByte

subByte :: Data Word8 -> Data Word8
subByte x = sbox' !! i2n x
```

The shape of the vector `State` is $[4, 4]$, which is irrelevant for mapping. Instead, we map subByte over each value. Keep in mind that a Feldspar vector is defined by a shape and a list of values. The shape specifies the dimensions of data given as a one-dimensional lists of values. Thus, `map` can ignore the shape, as it is possible to simply map over the list of values due to the fact that this operation does not change the shape of the provided vector.

Note that the operation `i2n`, which is used in `subByte`, turns a value into an integer, which is needed for indexing into the given vector. The vector indexing operator is `!!`.

The function `sbox'` below turns the provided Feldspar list of values into an indexed pull vector. Furthermore, the built-in function `value` transforms a Haskell list of values of type `[Data a]` into a Feldspar vector of type `Data [a]`.

```
sbox' :: Pull1 Word8
sbox' = indexed1 256 (\i -> sbox ! i)

sbox :: Data [Word8]
sbox = value [
            0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
            -- values omitted
            0x54, 0xbb, 0x16 ]
```

`Pull1` represents a one-dimensional pull vector. The function `indexed1` creates such a one-dimensional pull vector. In the λ-expression, the operator `!` is used for indexing into a list.

5 When describing Feldspar code, we may use the terms *array* and *vector* interchangeably.

The `shiftRows` operation is implemented in a more esoteric manner. The helper function `perm` modifies the shape of the vector.[6]

```
shiftRows :: State -> State
shiftRows v = backpermute (extent v) perm v
    where perm (Z :. row :. col) = Z :. row :. ((col + row) ‘mod‘ cols)
          cols = 4
```

In Feldspar, the `mixColumns` operation is much less straightforward than it is in Haskell:

```
mixColumns st = transpose $ arrToPull (Z :. 4 :. 4) abcd
    where [r0, r1, r2, r3] = map (mixColumn . colN st) [0, 1, 2, 3]
          [a, b, c, d]     = map thawPull1 [r0, r1, r2, r3]

          ab, cd           :: Data [Word8]
          [ab, cd]         = map freezePush1 [a +=+ b, c +=+ d]

          abcd             :: Data [Word8]
          abcd             = freezePush1 $ (thawPull1 ab) +=+ (thawPull1 cd)
```

Some comments may seem in order. In the *where*-clause, we first extract columns 0 to 3 from the current state, using the helper function `colN`. This function is not shown below. Applying `mixColumn` to each column results in a list of vectors `[r0, r1, r2, r3]`. However, our state is defined as an array, which requires a one-dimensional list of values. Therefore, we first turn vectors r_i into pull vectors via `thawPull1`, then concatenate the first and second as well as the third and fourth such vector. This step is repeated once, which leads to one one-dimensional vector `abcd`. We use those values as input for the function `arrToPull`, which creates a pull vector. After transposing, we get a valid representation of our state, i.e. another 4-by-4 pull array.

Some of the used functions need an explanation: `arrToPull` restores a vector from memory, `thawPull1` restores a vector and its shape from memory, `feezePush1` stores a one-dimensional vector and its shape to memory, and, finally, the operator +=+ concatenates a vector along the outer dimension. The operator +=+ expects both vectors to be of identical length, but Feldspar does not check that the two lengths are indeed equal.

The function call `arrToPull (Z :. 4 :. 4) abcd` is noteworthy as the syntax for specifying the shape, which is given by the first argument, has been taken from the Haskell library Repa [16]. In `mixColumns` we encounter a shape specification with concrete values, while we saw an abstract invocation of a shape in `shiftRows` above.

The next function is `mixColumn`:

```
mixColumn :: Data [Word8] -> Data [Word8]
mixColumn row = fromList [b0, b1, b2, b3]
    where b0,  b1, b2, b3 :: Data Word8
          b0 = (x2 a0) ‘xor‘ (x3 a1) ‘xor‘     a2  ‘xor‘     a3
          b1 =     a0  ‘xor‘ (x2 a1) ‘xor‘ (x3 a2) ‘xor‘     a3
          b2 =     a0  ‘xor‘     a1  ‘xor‘ (x2 a2) ‘xor‘ (x3 a3)
          b3 = (x3 a0) ‘xor‘     a1  ‘xor‘     a2  ‘xor‘ (x2 a3)
          a0 = row ! 0
          a1 = row ! 1
          a2 = row ! 2
          a3 = row ! 3
```

The bit operations in `mixColumn` are straightforward. However, note that the built-in operation `fromList` is required in order to create a Feldspar vector from a Haskell list. This function could be defined as a matrix multiplication

6 Replacing `cols` with 4 in the following code fragment leads to an obscure compiler error.

on finite fields, but in order to be consistent with the other implementations of AES, we, again, opt for hardcoding multiplications as table lookups. The functions `x2` and `x3` look up entries in tables that contains precomputed values of the corresponding multiplications.

Lastly, we would like to draw attention to the fact that Feldspar does not know enumerations like `[1 .. n]`, which are a common occurrence in Haskell source code. In `mixColumns`, we therefore need to specify the required list as `[0, 1, 2, 3]`.

Lastly, the implementation of the function `addRoundKey` is, again, very concise:

```
addRoundKey :: Key -> State -> State
addRoundKey = zipWith xor
```

This concision is due to `zipWith` being able to operate on the provided values of the state, leaving its shape unchanged. Thus, we *xor* each byte of the key with its corresponding byte in the given state in one pass.

6.4 C

6.4.1 *C, handwritten*

There are several open-source implementations of AES written in C available. We chose an existing implementation of AES that aimed at readability and clarity as the basis of our work.[7] Considering that the C source code the Feldspar compiler aims to generate is rather straightforward, it seemed adequate to compare it with a C implementation that is straightforward and readable, instead of an optimized one that is potentially convoluted.

In our implementation of AES in C, state is represented as a 2-dimensional array of unsigned characters:

```
unsigned char state[4][4];
```

In C, defining a function that takes a function as a parameter is cumbersome, and would not normally be done by C programmers anyway. Thus, we decided to define the execution of an AES main round by successively applying the following functions to a state.

```
SubBytes(state);
ShiftRows(state);
MixColumns(state);
AddRoundKey(state, key);
```

Keep in mind that in the C code below updates happen in place, i.e. memory locations are assigned new values. Previously, we have only seen code written in functional languages, where functions return values that were computed based on the input, but which left the input itself unmodified.

First, we show `SubBytes`, which performs a substitution of all bytes in the array that represents state with their corresponding values in the S-box:

```
void SubBytes(unsigned char state[][4])
{
  state[0][0] = sbox[state[0][0]];
  // code omitted
  state[3][3] = sbox[state[3][3]];
}
```

7 The C code in this subsection has been rewritten, based on the AES implementation of Brad Conte, which he released into the public domain, available at http://bradconte.com/aes_c (accessed February 1, 2016).

`ShiftRows` is likewise straightforward:

```
void ShiftRows(unsigned char state[][4])
{
  int t;

  t          = state[1][0];
  state[1][0] = state[1][1];
  state[1][1] = state[1][2];
  state[1][2] = state[1][3];
  state[1][3] = t;

  t          = state[2][0];
  state[2][0] = state[2][2];
  state[2][2] = t;
  t          = state[2][1];
  state[2][1] = state[2][3];
  state[2][3] = t;

  t          = state[3][0];
  state[3][0] = state[3][3];
  state[3][3] = state[3][2];
  state[3][2] = state[3][1];
  state[3][1] = t;
}
```

Note that temporary variables are necessary in `ShiftRows` in order to store values that occupy memory locations that need to be overwritten.

In `MixColumns` we see a direct *analogon* to the bit manipulation done in the three programming languages we previously had a look at:

```
void MixColumns(unsigned char state[][4])
{
  unsigned char v0, v1, v2, v3; // temporary values

  v0          = state[0][0];
  v1          = state[1][0];
  v2          = state[2][0];
  v3          = state[3][0];
  state[0][0]  = mul2[v0] ^ mul3[v1] ^ v2       ^ v3;
  state[1][0]  = v0       ^ mul2[v1] ^ mul3[v2] ^ v3;
  state[2][0]  = v0       ^ v1       ^ mul2[v2] ^ mul3[v3];
  state[3][0]  = mul3[v0] ^ v1       ^ v2       ^ mul2[v3];
  // columns 2 to 4 omitted
}
```

Note again the need for temporary variables in the function definition above. This is a consequence of updating values in place. Multiplication is implemented as a table lookup, using precomputed values. Concretely, the arrays `mul2` and `mul3` contain precomputed values for multiplication with 2 and 3, respectively.

Lastly, `AddRoundKey` is a simple *xor*ing of each byte in the state array with the corresponding byte in the array storing the round key.

```
void AddRoundKey(unsigned char state[][4]
                ,unsigned char key[]     )
{
   state[0][0] ^= key[0];
   state[0][1] ^= key[1];
   state[0][2] ^= key[2];
   state[0][3] ^= key[3];
  // code omitted
}
```

The C code we have seen is generally rather simple, if not elegant. In the following, we will look at the C code Feldspar generates.

6.4.2 *C, generated by Feldspar*

In our examples, C generated by Feldspar was generally rather verbose. In fact, it was so verbose that it seemed unreasonable to even include all of it in the appendix, as some functions expanded to hundreds of lines of code. In the following, we can therefore only present fragments. Below, we show the generated C code of the four functions that are called by `aesMainRound`. Note that compiling that function in its entirety would lead to different code, due to fusion. We present the various functions in isolation since this approach makes the code more easily understandable. Keep in mind that in the current version of Feldspar, when compiling a Feldspar function to C, the resulting function is named `test`.

One important note is that in the AES standard, substitutions are defined on a two-dimensional vector. Likewise, in the original Feldspar code, state is modeled as a two-dimensional vector. In the generated C code, however, those vectors are flattened into one-dimensional C arrays. Instead of indexing rows and columns separately, a position argument is computed that determines the corresponding element in a one-dimensional vector.

We start with the C code generated for `SubBytes`:

```
void test(struct s_2_arr_unsignedS32_arr_unsignedS8 * v0,
          struct s_2_arr_unsignedS32_arr_unsignedS8 * out)
{
  uint32_t v5;
  uint32_t v6;
  struct array * v7 = NULL;
  uint32_t len8;

  v5 = at(uint32_t,(*v0).member1,0);
  v6 = at(uint32_t,(*v0).member1,1);
  v7 = initArray(v7, sizeof(uint8_t), 256);  // sbox
  at(uint8_t,v7,0) = 99;
  // 255 lines of assignments omitted

  (*out).member1 = setLength((*out).member1, sizeof(uint32_t), 2);
  at(uint32_t,(*out).member1,0) = v5;
  at(uint32_t,(*out).member1,1) = v6;
  len8 = (v6 * v5);

  (*out).member2 = initArray((*out).member2, sizeof(uint8_t), len8);
  for (uint32_t v4 = 0; v4 < len8; v4 += 1)
  {
    at(uint8_t,(*out).member2,v4) = at(
      uint8_t,v7,((uint32_t)(at(
        uint8_t,(*v0).member2,(((v4 / v5) * v5) + (v4 % v5))))));
  }
  freeArray(v7);
}
```

In the code above, one first notices that even though the original Feldspar function had only one argument, the generated C function takes two arguments. Each is a struct with a pointer. One struct specifies the input, the other the output. Those structs contain the array dimensions — note the values `v5` and `v6` in the code below, which are first taken from the input struct and later on reused for the output struct. They are also required for computing the length `len8`, which is used as a boundary check for the for loop that assigns S-box values to the output struct. Also note that the S-box is inlined and reconstructed by individual assignments. A definition of those structs is not emitted by the Feldspar compiler. Similarly, one may notice functions like `initArray()`, `at()`, and `freeArray()`, whose definition is not provided either. The nascent RAW-Feldspar, whose prefix is a shorthand for *resource aware* may address this as it only compiles entire programs, which is not possible in the current version, as `main` inhabits the Haskell IO monad, which is not part of Feldspar.[8]

In the original Feldspar code, we constructed a vector as a function, which performs the substitution of values via looking up their corresponding entries in an S-box. In the code we have just seen, the initial S-box is unrolled, and each entry gets its value assigned separately. We list one such assignment, but omit the remaining 255. The substitution itself takes place in the for loop at the end.

The C code generated for `ShiftRows` is only represented by a fragment below.

```
void test(struct s_2_arr_unsignedS32_arr_unsignedS8 * v0,
          struct s_2_arr_unsignedS32_arr_unsignedS8 * out)
{
  // code omitted

  len8 = (v6 * v5);
  (*out).member2 = initArray((*out).member2, sizeof(uint8_t), len8);

  for (uint32_t v4 = 0; v4 < len8; v4 += 1)
  {
    v7 = (v4 / v5);
    at(uint8_t,(*out).member2,v4) = at(
      uint8_t,(*v0).member2,((v7 * v5) + (((v4 % v5) + v7) % 4)));
  }
}
```

In the code above, the for loop performs the cyclical rotation of rows by traversing the values of the state array one by one. The computation of the position of the values in `v0` is difficult to untangle, but may become clearer after mentally substituting some of the variables with known values. The variable `len8` contains the result of multiplying `v6` and `v5`, which are both equal to 4. Note that in the Feldspar source code, the number of rows and columns are specified when creating a state, as this requires a shape argument, which is `(Z :. 4 :. 4)`. In the generated C code, those values are, however, not directly reflected. Instead, the variables `v4` and `v5` are used to compute position values. In the for loop, `v4` is a running index that takes on values from 0 to 15, inclusive. Values 0 to 3 specify the first row in the state, values 4 to 7 the second row, values 8 to 11 the third row, and, finally, values 12 to 15 the fourth row. As a concrete example, let us compute the value of the output state `state_out` of a particular field, based on the input state `state_in`. The variable `v4` is referred to as `i`.

The equation is:

```
state_out[i] := state_in[ i + (((i % 4) + i/4) % 4) ]
```

Now, let us compute one value:

```
state_out[8] := state_in[ 8 + ((2 + 0) % 4) ] = state_in[10]
```

This is as expected, based on the description of the procedure `ShiftRows` in the AES standard. Concretely, this assignment means that the third value

8 RAW-Feldspar is available at https://github.com/emilaxelsson/raw-feldspar (accessed February 10, 2016).

in the third row of the input state (10) is put in the first position of the third row in the output state (8).

Unfortunately, `MixColumns` leads to an inordinate amount of C code being generated. In the original Feldspar code we performed several array transformations. Those are all unrolled in the generated C code. Thus, we only show a small fragment that reflects part of the *xor* operations in the Feldspar source code. The first block shows multiplication via table lookups and subsequent *xor*ing of values. Multiplication by 1 is simply a reassignment of the unchanged old value to the corresponding position in the output struct.

```
void test(struct s_2_arr_unsignedS32_arr_unsignedS8 * v0,
          struct s_2_arr_unsignedS32_arr_unsignedS8 * out)
{
  // code omitted

  // XOR operations:
  v12315 = (((at(uint8_t,v12417,((uint32_t)(at(uint8_t,v12314,0))))
    ^ at(uint8_t,v12418,((uint32_t)(at(uint8_t,v12314,1)))))
    ^ at(uint8_t,v12314,2))
    ^ at(uint8_t,v12314,3));
  v12320 = setLength(v12320, sizeof(uint8_t), 4);
  at(uint8_t,v12320,0) = v12413;
  at(uint8_t,v12320,1) = v12414;
  at(uint8_t,v12320,2) = v12415;
  at(uint8_t,v12320,3) = v12416;

  //code omitted
}
```

Lastly, in the C code generated for the Feldspar function `addRoundKey` we again see the addition of an output struct. The original `zipWith xor` is turned into a for loop, in which bytes are *xor*ed one by one and assigned to the output struct.

```
void test(struct s_2_arr_unsignedS32_arr_unsignedS8 * v0,
          struct s_2_arr_unsignedS32_arr_unsignedS8 * v1,
          struct s_2_arr_unsignedS32_arr_unsignedS8 * out)
{
  // code omitted

  (*out).member2 = initArray((*out).member2, sizeof(uint8_t), len14);
  for (uint32_t v5 = 0; v5 < len14; v5 += 1)
  {
    v10 = (v5 / v6);
    v11 = (v5 % v6);
    at(uint8_t,(*out).member2,v5) = (at(
      uint8_t,(*v0).member2,((v10 * v12) + v11)) ^ at(
        uint8_t,(*v1).member2,((v10 * v13) + v11)));
  }
}
```

7 DISCUSSION

Our subsequent discussion of the strengths and weaknesses of Feldspar has to be put in perspective first. Thus, we stress that Feldspar was designed for the digital signal processing domain. In this study, we explored its applicability for implementing cryptographic block ciphers, which is a different domain. It therefore is arguably no surprise that we encountered some difficulties. On the other hand, the mere fact that we were able to implement the main part of AES is a demonstration of the flexibility of Feldspar. Feldspar may lack some of the convenience provided by Haskell or Cryptol, but the code presented in the appendix nonetheless correctly implements the main round of AES-128.

The original intention of comparing C source code generated by Cryptol and Feldspar, both in terms of performance and on a syntactic level, with hand-written C did not come to fruition. The current version of Cryptol lacks a C backend, which may be added in the future. Furthermore, C code as generated by Feldspar requires some attention in order to turn it into executable C code. In section 2 we have seen that arguments in Feldspar functions are turned into C structs whose definitions are not provided. Furthermore, several helper functions are called, but not defined either.

The full listing of the C code generated from Feldspar, with some elisions, can be found in the appendix.

7.1 Drawbacks

We will discuss C code as generated by Feldspar in greater detail in the next subsection. In general, though, it has to be stressed that the current version does *not* produce C code that is ready to use, as it contains references to structs it does not define, or makes use of type synonyms as well as helper functions like `initArray()` and `freeArray()` that are not part of its output, as discussed in section 6.4.2. Both issues may be addressed by RAW-Feldspar.

Both in Cryptol and Haskell, manipulating AES state, as it is modeled in our source code, is straightforward, as the chosen representation is a nested list (Haskell) or a nested sequence (Cryptol), which is just a terminological difference. However, Feldspar does not know a comparable data structure. In fact, Haskell lists have to be transformed to Feldspar vectors. In both Cryptol and Haskell, extracting or manipulating rows of the nested data structures these languages use is straightforward. On the other hand, Feldspar lacks this convenience. As we have seen, a Feldspar user needs to extract rows and columns, either with custom written code as we have done with a helper function called `colN`, or by using *slices*, which extract part of a Feldspar vector.

A possible workaround in Feldspar would be the use of nested vectors. By indexing into such a vector, one could extract a vector representing a row. This would only move the necessary plumbing elsewhere, however. For instance, for `subBytes`, such a nested vector would need to be flattened, or that function rewritten. However, note that a nested vector would lead to the generation of substantially more C code, as each vector will have a representation as its own struct in C. We have encountered this problem before in section 6.4.2, when we were facing substantial amounts of assignment statements, which were due to vector definitions being unrolled. This alone is an impediment towards using Feldspar for practical code generations, as it can easily make the output difficult to read.

We have seen that sequence comprehensions in Cryptol, as well as list comprehensions in Haskell can lead to concise, easily readable code. There is no corresponding construct in Feldspar, but it may be potentially useful, particularly for nested lists or sequences, to use the Haskell and Cryptol terminology, respectively. There may be other uses for comprehension syntax, for instance for nested vectors, or there may even be possible uses for shapes. For instance, a shape could be specified via variables, in order to process all values of an array.

The absence of Haskell-like enumerations like `[1 .. n]` is a minor inconvenience, as seen in section 6.3. In our Feldspar code, we never encountered an n larger than 3, so manually specifying all values of such a list was trivial. However, in other contexts, the absence of enumerations may make some programs cumbersome to write.

7.2 Generated C output

C code, as it is generated by Feldspar, leads to possibly surprising results. First and foremost, the representation of Feldspar arrays as structs, instead of C arrays, is noticeable. Take the following header of generated C code, which represents the `SubBytes` function:

```
void test(struct s_2_arr_unsignedS32_arr_unsignedS8 * v0,
          struct s_2_arr_unsignedS32_arr_unsignedS8 * out)
```

As the type signature of the corresponding Feldspar function indicates, this function takes a `State` as input and produces a `State` as output:

```
subBytes :: State -> State
```

However, in the generated C code, the function `test()` takes two arguments as its input. The first is a pointer to a struct with input values, while the second is a pointer to the output this function generates. A closer look at the beginning of section 6.4.2, where we had a look at `SubBytes`, reveals the symmetry in the generated code. First, relevant values are extracted from the struct `v0` in the function `test`. After code accessing `v0` has been executed, the corresponding fields in the output struct, `out`, are set.

The Feldspar C backend inlines exhaustively. As we have previously hinted at near the end of section 6.4.2, on page 22, and which the generated code in the appendix shows more clearly, this leads to great verbosity. For instance, the array holding the values of the Rijndael S-box is filled one element per line when compiling the Feldspar function `subBytes` to C:

```
v7 = initArray(v7, sizeof(uint8_t), 256);
at(uint8_t,v7,0) = 99;
// remaining 255 values omitted
```

When compiling `sbox` or `sbox'`, the same result appears. However, it is not clear why Feldspar compiles the S-box to an array that is initialized with all values at once when compiling `aesMainRound`:

```
uint8_t v12438[] = {99,
                    124,
// remaining 254 values omitted
}
```

As was pointed out earlier, Feldspar is a moving target. In fact, RAW-Feldspar, developed in a separate branch, is intended to alleviate some of the issues pointed out in this subsection. For instance, RAW-Feldspar no longer initializes arrays one value at a time, and furthermore no longer creates separate structs for the input and output of functions, even when the original Feldspar function only takes one value as an argument.

7.3 General benefits of using Feldspar

Feldspar shares the benefits of its host language for increasing programmer productivity, such as expressive, concise syntax. We have seen several instances of using the higher-order function `map`, and one instance of `zipWith`. Similar or identical language constructs are available in Cryptol and Haskell as well. Note that Feldspar is not using the Haskell equivalences of said functions, but instead has its own implementations, suited to its data types. In comparison, C code is generally more verbose. As an example, let us recall the Feldspar definition of `addRoundKey`:

```
addRoundKey :: Key -> State -> State
addRoundKey = zipWith xor
```

In C, the equivalent function is as follows:

```
void AddRoundKey(unsigned char state[][4]
                ,unsigned char key[]     )
{
   state[0][0] ^= key[0];
   state[0][1] ^= key[1];
   state[0][2] ^= key[2];
   state[0][3] ^= key[3];

   state[1][0] ^= key[4];
   state[1][1] ^= key[5];
   state[1][2] ^= key[6];
   state[1][3] ^= key[7];

   state[2][0] ^= key[8];
   state[2][1] ^= key[9];
   state[2][2] ^= key[10];
   state[2][3] ^= key[11];

   state[3][0] ^= key[12];
   state[3][1] ^= key[13];
   state[3][2] ^= key[14];
   state[3][3] ^= key[15];
}
```

It arguably goes without saying that the Feldspar code is much faster to write and easier to debug. On the other hand, each single bit operation in the corresponding C code is a potential source of errors.

Related to the issue of productivity is the fact that C requires manual memory management, which commonly leads to programmer errors. C code generated with Feldspar, though, takes care of this problem, so that programmers could concentrate on correctly implementing a Feldspar program, without the burden of manual memory management. Manual memory management was not an issue in the C code we provided, due to in-place updates. However, this may not always be an option. Thus, the automation via Feldspar may come in handy. We have seen this expressed in the functions `initArray()` and `freeArray()` in the generated C code.

Furthermore, by using Haskell's type system, Feldspar is able to provide guarantees much stronger than the rather rudimentary ones of the type system of C. This means that a Haskell compiler is able to detect many more programming errors than a C compiler would, thus avoiding the issue that C source code can be difficult to debug. As a simple example, take out-of-bounds memory accesses. In the C function `AddRoundKey` above, for instance, a programmer could access `key[16]`, even though the corresponding array does not extent to that memory location, and would retrieve whatever value happened to be stored there.

The observations just highlighted are relevant for practice. Whenever C code is required, one could either write C by hand, or have it generated. The latter is possible with Feldspar. In fact, it would be a rather elegant way to create, assuming the Feldspar C backend is correct, more robust C code than would be possible manually. This is due to the fact that before Feldspar emits C code, the Feldspar source code has to satisfy the Haskell compiler first.

As can be seen from the provided output of the Feldspar C backend, generated C code is not necessarily easy to read, which is an issue we already covered. However, one could think of situations in which either individual functions are written in Feldspar, compiled to C, and used in a larger C program, for the benefits outlines further above. Or, compile a larger Feldspar program to C, and use the generated code instead of writing a larger C program by hand. An advantage of this is that the compiler could then perform various optimizations, like fusion. Of course, this may to less readable output. It may be feasible to approach software development this way when maintainability is only desired for the Feldspar source code, and not the generated C source code. This may not be an overly plausible scenario, though.

7.4 The Feldspar Ecosystem

Feldspar is work in progress, and developed by only a small number of people. Thus, it is inevitable that it lacks the polish and support one may find for more established programming languages. It seems safe to diagnose that Feldspar faces two hurdles to wider adoption outside academia. One is that its host language, Haskell, is still a niche programming language. Anyone wanting to explore Feldspar for the domains it was originally intended for may thus face the hurdle of having to familiarize themselves with a possibly unfamiliar programming paradigm. This problem will be hard to resolve, considering the lack of adoption of functional programming languages in industry.

On the other hand, even programmers familiar with other functional programming languages may encounter difficulties. One stumbling block is the lack of documentation. Due to changes to Feldspar, the main tutorial [3] is partly out of date, and the tutorial that is part of the official Feldspar repository is not nearly as comprehensive. This is exacerbated by the relative absence of programming examples, including a folder with examples that is part of the official repository.[9] Learning from examples is, due to their relative absence, therefore rather difficult.

Studying the source code of relevant Feldspar modules has limitations due to the sparsity of explanatory comments of functions, and the absence of examples. As was noted in section 6.3 on page 17, part of the Feldspar vector library has been inspired by Repa, a library for parallel array computations in Haskell [16]. In order to figure out how to use parts of the Feldspar vector library, we needed to resort to the documentation of the Haskell parallel array library Repa. In order to make this connection, one either needs to know that Feldspar's vector library has been inspired by Repa, or be familiar with Repa and notice the similarity between that Haskell library and the notation

9 The official Github repository of Feldspar includes code examples in this directory: `https://github.com/Feldspar/feldspar-language/tree/master/examples` (accessed January 15, 2016).

that Feldspar uses in some parts of its vector library. There are limitations to referring to Repa, though. While Repa is fairly well-documented, Feldspar does not fully replicate it, meaning that due to differences there is only a partial correspondence between Repa and Feldspar's vector library, which is a potential source of confusion. Adding some examples to relevant parts of the Feldspar source code would arguably solve this problem.

8 FUTURE WORK

In this study we explored the suitability of Feldspar for cryptographic block ciphers, based on AES. It would certainly be interesting to implement further block ciphers in Feldspar, but not necessarily following a similar comparative approach as we have done. Instead, comparing handwritten C code with C code Feldspar generates promises to provide further insights regarding possible improvements of Feldspar's C backend.

Particularly in the implementation of the `shiftRow` operation in Feldspar, we encountered some difficulties, particularly in comparison to Cryptol and Haskell. Limited support for pattern matching on vectors would have made work in Feldspar more convenient. It is easy to imagine other situations in which pattern matching would pay off by avoiding what could rightly be called boilerplate code, which has to be written as a workaround.

In our Cryptol and Haskell code, sequence and list comprehensions, respectively, led to rather concise code. Feldspar lacks such a construct. In combination with nested vectors, or possibly even shapes, a language construct for comprehensions could therefore be a useful addition to Feldspar.

Related to our work on block ciphers would be a project on implementing stream ciphers — stream ciphers are symmetric key ciphers that *xor* the plaintext with a keystream. Feldspar was originally designed as a digital signal processing languages, where signals are likewise streams. In fact, there has been some recent work on extending the stream processing capabilities of Feldspar [1], so that an implementation of stream ciphers might be a worthwhile next step.

Lastly, let us remark that it is difficult to recommend particular modifications to the Feldspar C backend, precisely since a current work-in-progress version, the aforementioned RAW-Feldspar is intended to address some of the issues we highlighted.

9 ACKNOWLEDGEMENTS

I am grateful for the open and supportive environment the functional programming research group at Chalmers University of Technology provided. First and foremost, I would like to thank Michał Pałka and Mary Sheeran for suggesting this topic, Michał for supervising this project, for his guidance and valuable feedback, and Mary for agreeing to be the examiner.

Working with a research programming language poses a significant challenge, primarily because it is difficult to get support, due to a limited user base, and the fact that the language may change faster than the documentation associated with it. I am therefore grateful for the support and assistance I received of what amounted to virtually the entire current Feldspar implementor community. In alphabetical order, by surname, I therefore extend my thanks to Markus Aronsson for helping me getting to grips with the Feldspar vector library, Emil Axelsson for providing concrete help when the types were guiding me but the corresponding functions failed to properly materialize, Anders Persson for help with the arcane Feldspar function `backpermute`, and Josef Svenningsson for an insightful discussion on implementing AES and sharing his views on the motivation behind some of the decisions behind the current implementation of the Feldspar vector library.

REFERENCES

[1] Aronsson, Markus, Emil Axelsson, and Mary Sheeran. "Stream Processing for Embedded Domain Specific Languages." In Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages, p. 8. ACM, 2014.

[2] Axelsson, Emil, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. "Feldspar: A domain specific language for digital signal processing algorithms." In Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on, pp. 169-178. IEEE, 2010.

[3] Axelsson, Emil, Anders Persson, Mary Sheeran, Josef Svenningsson, and Gergely Dévai. "A Tutorial on Programming in Feldspar." (2011).

[4] Buxton, John N., and Brian Randell, eds. Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee. NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[5] Claessen, Koen, and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs." Acm sigplan notices 46, no. 4 (2011): 53-64.

[6] Courtois, Nicolas T., and Josef Pieprzyk. "Cryptanalysis of block ciphers with overdefined systems of equations." In Advances in Cryptology – ASIACRYPT 2002, pp. 267-287. Springer Berlin Heidelberg, 2002.

[7] Daemen, Joan, and Vincent Rijmen. "The block cipher Rijndael." In Smart Card Research and Applications, pp. 277-284. Springer Berlin Heidelberg, 1998. Harvard

[8] Dennis, Jack B. On the design and specification of a common base language. No. MAC-TR-101. Massachusetts Institute of Technology, Project MAC, 1972.

[9] Dennis, Jack B. "First version of a data flow procedure language." In Programming Symposium, pp. 362-376. Springer Berlin Heidelberg, 1974.

[10] De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." In Tools and Algorithms for the Construction and Analysis of Systems, pp. 337-340. Springer Berlin Heidelberg, 2008.

[11] Dévai, Gergely, Máté Tejfel, Zoltán Gera, Gábor Páli, Gyula Nagy, Zoltán Horváth, Emil Axelsson et al. "Efficient code generation from the high-level domain-specific language Feldspar for DSPs." In ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems, pp. 455-481. 2010.

[12] Dworkin, Morris. Recommendation for block cipher modes of operation. methods and techniques. No. NIST-SP-800-38A. National Institute of Standards and Technology, 2001.

[13] Erkök, Levent, Dylan McNamee, Joe Kiniry, Iavor Diatchki, and John Launchbury. Programming Cryptol. Galois, Inc. 2015. http://www.cryptol.net/files/ProgrammingCryptol.pdf (accessed 10 January 2016).

[14] Ferguson, Niels, Bruce Schneier, and Tadayoshi Kohno. Cryptography engineering. John Wiley & Sons, 2010.

[15] Katz, Jonathan, and Yehuda Lindell. Introduction to modern cryptography: principles and protocols. CRC press, 2007.

[16] Keller, Gabriele, Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. "Regular, shape-polymorphic, parallel arrays in Haskell." In ACM Sigplan Notices, vol. 45, no. 9, pp. 261-272. ACM, 2010.

[17] Lewis, Jeffrey R., and Brad Martin. "Cryptol: High assurance, retargetable crypto development and validation." In Military Communications Conference, 2003. MILCOM'03. 2003 IEEE, vol. 2, pp. 820-825. IEEE, 2003.

[18] Lewis, Jeff. "Cryptol: specification, implementation and verification of high-grade cryptographic applications." In Proceedings of the 2007 ACM workshop on Formal methods in security engineering, pp. 41-41. ACM, 2007.

[19] NIST FIPS. "197: Advanced encryption standard (AES)." Federal Information Processing Standards Publication 197 (2001): 441-0311.

[20] Shannon, Claude E. "Communication theory of secrecy systems*." Bell system technical journal 28, no. 4 (1949): 656-715.

A AES IMPLEMENTATIONS

A.1 Cryptol

The code below was written in Cryptol 2.3.0. Again, note that it is largely based on the AES implementation provided in the official Cryptol tutorial [13]. The implementation of the `ShiftRows` and `MixColumns` steps is entirely our own, however.

```
////////////////////////////////////////////////////////////////////////////////
//
// AES-128: Main Round
//
////////////////////////////////////////////////////////////////////////////////

type State = [4][4][8]

AESMainRound : (State, State) -> State
AESMainRound (rk, s) = AddRoundKey (rk, MixColumns (ShiftRows (SubBytes s)))


////////////////////////////////////////////////////////////////////////////////
// SubBytes
////////////////////////////////////////////////////////////////////////////////

SubBytes : State -> State
SubBytes state = [ [ sbox @ b | b <- row ] | row <- state ]

sbox : [256][8]
sbox = [ 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
         0x2b, 0xfe, 0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59,
         0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7,
         0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
         0x71, 0xd8, 0x31, 0x15, 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05,
         0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09, 0x83,
         0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
         0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
         0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, 0xd0, 0xef, 0xaa,
         0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c,
         0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc,
         0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c, 0x13, 0xec,
         0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
         0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee,
         0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49,
         0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
         0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4,
         0xea, 0x65, 0x7a, 0xae, 0x08, 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6,
         0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, 0x70,
         0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9,
         0x86, 0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e,
         0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, 0x8c, 0xa1,
         0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
         0x54, 0xbb, 0x16 ]


////////////////////////////////////////////////////////////////////////////////
// ShiftRows
////////////////////////////////////////////////////////////////////////////////

ShiftRows : State -> State
ShiftRows [ row1 ,
            row2 ,
            row3 ,
            row4 ] = [ row1 <<< 0 ,
                       row2 <<< 1 ,
                       row3 <<< 2 ,
                       row4 <<< 3 ]




////////////////////////////////////////////////////////////////////////////////
// MixColumns
////////////////////////////////////////////////////////////////////////////////

MixColumns : State -> State
MixColumns [ [a1, a2, a3, a4] ,
             [b1, b2, b3, b4] ,
             [c1, c2, c3, c4] ,
             [d1, d2, d3, d4] ] = transpose [ MixCol [ a1, b1, c1, d1 ] ,
                                              MixCol [ a2, b2, c2, d2 ] ,
                                              MixCol [ a3, b3, c3, d3 ] ,
                                              MixCol [ a4, b4, c4, d4 ] ]

MixCol : [4][8] -> [4][8]
MixCol [a0, a1, a2, a3] = [b0, b1, b2, b3]
    where b0 = mult2 @ a0 ^ mult3 @ a1 ^         a2 ^         a3
          b1 =         a0 ^ mult2 @ a1 ^ mult3 @ a2 ^         a3
          b2 =         a0 ^         a1 ^ mult2 @ a2 ^ mult3 @ a3
          b3 = mult3 @ a0 ^         a1 ^         a2 ^ mult2 @ a3

mult2 : [256][8]
mult2 = [ 0x00,0x02,0x04,0x06,0x08,0x0a,0x0c,0x0e,0x10,0x12,0x14,0x16,0x18,
          0x1a,0x1c,0x1e,0x20,0x22,0x24,0x26,0x28,0x2a,0x2c,0x2e,0x30,0x32,
          0x34,0x36,0x38,0x3a,0x3c,0x3e,0x40,0x42,0x44,0x46,0x48,0x4a,0x4c,
          0x4e,0x50,0x52,0x54,0x56,0x58,0x5a,0x5c,0x5e,0x60,0x62,0x64,0x66,
          0x68,0x6a,0x6c,0x6e,0x70,0x72,0x74,0x76,0x78,0x7a,0x7c,0x7e,0x80,
          0x82,0x84,0x86,0x88,0x8a,0x8c,0x8e,0x90,0x92,0x94,0x96,0x98,0x9a,
          0x9c,0x9e,0xa0,0xa2,0xa4,0xa6,0xa8,0xaa,0xac,0xae,0xb0,0xb2,0xb4,
          0xb6,0xb8,0xba,0xbc,0xbe,0xc0,0xc2,0xc4,0xc6,0xc8,0xca,0xcc,0xce,
          0xd0,0xd2,0xd4,0xd6,0xd8,0xda,0xdc,0xde,0xe0,0xe2,0xe4,0xe6,0xe8,
          0xea,0xec,0xee,0xf0,0xf2,0xf4,0xf6,0xf8,0xfa,0xfc,0xfe,0x1b,0x19,
          0x1f,0x1d,0x13,0x11,0x17,0x15,0x0b,0x09,0x0f,0x0d,0x03,0x01,0x07,
          0x05,0x3b,0x39,0x3f,0x3d,0x33,0x31,0x37,0x35,0x2b,0x29,0x2f,0x2d,
          0x23,0x21,0x27,0x25,0x5b,0x59,0x5f,0x5d,0x53,0x51,0x57,0x55,0x4b,
          0x49,0x4f,0x4d,0x43,0x41,0x47,0x45,0x7b,0x79,0x7f,0x7d,0x73,0x71,
          0x77,0x75,0x6b,0x69,0x6f,0x6d,0x63,0x61,0x67,0x65,0x9b,0x99,0x9f,
          0x9d,0x93,0x91,0x97,0x95,0x8b,0x89,0x8f,0x8d,0x83,0x81,0x87,0x85,
          0xbb,0xb9,0xbf,0xbd,0xb3,0xb1,0xb7,0xb5,0xab,0xa9,0xaf,0xad,0xa3,
          0xa1,0xa7,0xa5,0xdb,0xd9,0xdf,0xdd,0xd3,0xd1,0xd7,0xd5,0xcb,0xc9,
          0xcf,0xcd,0xc3,0xc1,0xc7,0xc5,0xfb,0xf9,0xff,0xfd,0xf3,0xf1,0xf7,
          0xf5,0xeb,0xe9,0xef,0xed,0xe3,0xe1,0xe7,0xe5 ]

mult3 : [256][8]
mult3 = [ 0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,0x18,0x1b,0x1e,0x1d,0x14,
          0x17,0x12,0x11,0x30,0x33,0x36,0x35,0x3c,0x3f,0x3a,0x39,0x28,0x2b,
          0x2e,0x2d,0x24,0x27,0x22,0x21,0x60,0x63,0x66,0x65,0x6c,0x6f,0x6a,
          0x69,0x78,0x7b,0x7e,0x7d,0x74,0x77,0x72,0x71,0x50,0x53,0x56,0x55,
          0x5c,0x5f,0x5a,0x59,0x48,0x4b,0x4e,0x4d,0x44,0x47,0x42,0x41,0xc0,
          0xc3,0xc6,0xc5,0xcc,0xcf,0xca,0xc9,0xd8,0xdb,0xde,0xdd,0xd4,0xd7,
          0xd2,0xd1,0xf0,0xf3,0xf6,0xf5,0xfc,0xff,0xfa,0xf9,0xe8,0xeb,0xee,
          0xed,0xe4,0xe7,0xe2,0xe1,0xa0,0xa3,0xa6,0xa5,0xac,0xaf,0xaa,0xa9,
          0xb8,0xbb,0xbe,0xbd,0xb4,0xb7,0xb2,0xb1,0x90,0x93,0x96,0x95,0x9c,
          0x9f,0x9a,0x99,0x88,0x8b,0x8e,0x8d,0x84,0x87,0x82,0x81,0x9b,0x98,
          0x9d,0x9e,0x97,0x94,0x91,0x92,0x83,0x80,0x85,0x86,0x8f,0x8c,0x89,
          0x8a,0xab,0xa8,0xad,0xae,0xa7,0xa4,0xa1,0xa2,0xb3,0xb0,0xb5,0xb6,
          0xbf,0xbc,0xb9,0xba,0xfb,0xf8,0xfd,0xfe,0xf7,0xf4,0xf1,0xf2,0xe3,
          0xe0,0xe5,0xe6,0xef,0xec,0xe9,0xea,0xcb,0xc8,0xcd,0xce,0xc7,0xc4,
          0xc1,0xc2,0xd3,0xd0,0xd5,0xd6,0xdf,0xdc,0xd9,0xda,0x5b,0x58,0x5d,
          0x5e,0x57,0x54,0x51,0x52,0x43,0x40,0x45,0x46,0x4f,0x4c,0x49,0x4a,
          0x6b,0x68,0x6d,0x6e,0x67,0x64,0x61,0x62,0x73,0x70,0x75,0x76,0x7f,
          0x7c,0x79,0x7a,0x3b,0x38,0x3d,0x3e,0x37,0x34,0x31,0x32,0x23,0x20,
          0x25,0x26,0x2f,0x2c,0x29,0x2a,0x0b,0x08,0x0d,0x0e,0x07,0x04,0x01,
          0x02,0x13,0x10,0x15,0x16,0x1f,0x1c,0x19,0x1a ]



////////////////////////////////////////////////////////////////////////////////
// AddRoundKey
////////////////////////////////////////////////////////////////////////////////

AddRoundKey : (State, State) -> State
AddRoundKey (rk, s) = rk ^ s
```

A.2 Haskell

The Haskell code below was compiled with GHC 7.10.2. Considering that we do not make use of any extensions and exclusively rely on mainstream libraries, it should compile even in GHC versions much older than the one we used.

```
{-

AES 128: Main Round

-}

import Data.Array
import Data.Bits
import Data.List
import Data.Word

type State      = [[Word8]]


aesMainRound :: State -> State -> State
aesMainRound rk = addRoundKey rk . mixColumns . shiftRows . subBytes


--------------------------------------------------------------------------------
-- SubBytes
--------------------------------------------------------------------------------

subBytes :: State -> State
subBytes = map row
    where row = map subByte

subByte :: Word8 -> Word8
subByte x = sbox ! x

sbox :: Array Word8 Word8
sbox = array (0, 255) $ zip [0..] vals
    where vals = [
            0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
            -- values omitted
            0x54, 0xbb, 0x16 ]


--------------------------------------------------------------------------------
-- ShiftRows
--------------------------------------------------------------------------------

shiftRows :: State -> State
shiftRows [[a1, a2, a3, a4],
           [b1, b2, b3, b4],
           [c1, c2, c3, c4],
           [d1, d2, d3, d4]] = [[a1, a2, a3, a4],
                                [b2, b3, b4, b1],
                                [c3, c4, c1, c2],
                                [d4, d1, d2, d3]]


--------------------------------------------------------------------------------
-- MixColumns
--------------------------------------------------------------------------------
mixColumns :: State -> State
mixColumns [ [a1, a2, a3, a4] ,
             [b1, b2, b3, b4] ,
             [c1, c2, c3, c4] ,
             [d1, d2, d3, d4] ] = transpose [ mixColumn [ a1, b1, c1, d1 ] ,
                                              mixColumn [ a2, b2, c2, d2 ] ,
                                              mixColumn [ a3, b3, c3, d3 ] ,
                                              mixColumn [ a4, b4, c4, d4 ] ]

-- more concise:
-- mixColumns = transpose . map mixColumn . transpose

mixColumn :: [Word8] -> [Word8]
mixColumn [a0, a1, a2, a3] = [b0, b1, b2, b3]
    where b0 = mult2 ! a0 ‘xor‘ mult3 ! a1 ‘xor‘         a2 ‘xor‘         a3
          b1 =         a0 ‘xor‘ mult2 ! a1 ‘xor‘ mult3 ! a2 ‘xor‘         a3
          b2 =         a0 ‘xor‘         a1 ‘xor‘ mult2 ! a2 ‘xor‘ mult3 ! a3
          b3 = mult3 ! a0 ‘xor‘         a1 ‘xor‘         a2 ‘xor‘ mult2 ! a3

mult2 :: Array Word8 Word8
mult2 = array (0, 255) $ zip [0..] vals
    where vals = [
            0x00,0x02,0x04,0x06,0x08,0x0a,0x0c,0x0e,0x10,0x12,0x14,0x16,0x18,
            -- values omitted
            0xf5,0xeb,0xe9,0xef,0xed,0xe3,0xe1,0xe7,0xe5 ]

mult3 :: Array Word8 Word8
mult3 = array (0, 255) $ zip [0..] vals
    where vals = [
            0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,0x18,0x1b,0x1e,0x1d,0x14,
            -- values omitted
            0x02,0x13,0x10,0x15,0x16,0x1f,0x1c,0x19,0x1a ]


--------------------------------------------------------------------------------
-- AddRoundKey
--------------------------------------------------------------------------------
addRoundKey :: State -> State -> State
addRoundKey  = zipWith (zipWith (xor))
```

A.3 Feldspar

The code below was written with the Feldspar version that was current as of March 1, 2016. Note that at the time of writing the Feldspar version on Hackage significantly lags behind the official Feldspar repository on Github. In fact, the Hackage version cannot be built on GHC 7.10.

```
{-# LANGUAGE GADTs #-}

import Data.Word
import Prelude hiding (zip, map, take, drop, (++), (!!), mod, zipWith, splitAt)

import Feldspar
import Feldspar.Vector
import Feldspar.Compiler


type State =  Pull DIM2 (Data Word8)
type Key   = State

aesMainRound :: Key -> State -> State
aesMainRound k = addRoundKey k . mixColumns . shiftRows . subBytes


--------------------------------------------------------------------------------
-- SubBytes
--------------------------------------------------------------------------------

subBytes :: State -> State
subBytes = map subByte

subByte :: Data Word8 -> Data Word8
subByte x = sbox' !! i2n x

sbox' :: Pull1 Word8
sbox' = indexed1 256 (sbox !)

sbox :: Data [Word8]
sbox = value [
            0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
            -- values omitted
            0x54, 0xbb, 0x16 ]

shiftRows :: State -> State
shiftRows v = backpermute (extent v) perm v
    where perm (Z :. row :. col) = Z :. row :. ((col + row) `mod` cols)
          cols = 4


--------------------------------------------------------------------------------
-- MixColumns
--------------------------------------------------------------------------------

mixColumns :: State -> State
mixColumns st = transpose res
    where [r0, r1, r2, r3] = map (mixColumn . colN st) [0, 1, 2, 3]
          [a, b, c, d]     = map thawPull1 [r0, r1, r2, r3]

          ab, cd :: Data [Word8]
          [ab, cd] = map freezePush1 [a +=+ b, c +=+ d]

          abcd :: Data [Word8]
          abcd = freezePush1 $ (thawPull1 ab) +=+ (thawPull1 cd)

          res ::  State
          res =   arrToPull (Z :. 4 :. 4) abcd
```

```
colN :: State -> Data Length ->  Data [Word8]
colN st n = fromList $ map (\x -> st !: (Z :. x :. n)) [0, 1, 2, 3]


mixColumn :: Data [Word8] -> Data [Word8]
mixColumn row = fromList [b0, b1, b2, b3]
    where b0,  b1, b2, b3 :: Data Word8
          b0 = (x2 a0) ‘xor‘ (x3 a1) ‘xor‘     a2  ‘xor‘     a3
          b1 =     a0  ‘xor‘ (x2 a1) ‘xor‘ (x3 a2) ‘xor‘     a3
          b2 =     a0  ‘xor‘     a1  ‘xor‘ (x2 a2) ‘xor‘ (x3 a3)
          b3 = (x3 a0) ‘xor‘     a1  ‘xor‘     a2  ‘xor‘ (x2 a3)
          a0 = row ! 0
          a1 = row ! 1
          a2 = row ! 2
          a3 = row ! 3


x2 :: Data Word8 -> Data Word8
x2 x = mult2’ !! i2n x

mult2’ :: Pull1 Word8
mult2’ = indexed1 256 (\i -> mult2 ! i)

mult2 :: Data [Word8]
mult2 = value [
            0x00,0x02,0x04,0x06,0x08,0x0a,0x0c,0x0e,0x10,0x12,0x14,0x16,0x18,
            -- values omitted
            0xf5,0xeb,0xe9,0xef,0xed,0xe3,0xe1,0xe7,0xe5 ]

x3 :: Data Word8 -> Data Word8
x3 x = mult3’ !! i2n x

mult3’ :: Pull1 Word8
mult3’ = indexed1 256 (\i -> mult3 ! i)

mult3 :: Data [Word8]
mult3 = value [
            0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,0x18,0x1b,0x1e,0x1d,0x14,
            -- values omitted
            0x02,0x13,0x10,0x15,0x16,0x1f,0x1c,0x19,0x1a ]


--------------------------------------------------------------------------------
-- AddRoundKey
--------------------------------------------------------------------------------

addRoundKey :: Key -> State -> State
addRoundKey = zipWith xor
```

A.4 Feldspar: Alternative `MixColumns` definition

This alternative definition of `mixColumns` demonstrates the flexibility of Feldspar. The version below flattens the input into a one-dimensional array, which is subsequently split.

```
mixColumns :: State -> State
mixColumns st = transpose res
    where
        st’ = transpose st

        st’’ :: DPull DIM1 Word8
        st’’ = reshape (Z :. 16 ) st’      -- reshape into 1d array

        ab, cd :: DPull DIM1 Word8
        (ab, cd)  = halve st’’             -- or: splitAt 8 st’’

        ((a,b), (c,d)) = (halve ab, halve cd)

        a’, b’, c’, d’ :: DPull DIM1 Word8
        [a’, b’, c’, d’] = map (thawPull1 . mixColumn . fromPull) [a, b, c, d]

        ab’, cd’ :: Data [Word8]
        [ab’, cd’] = map freezePush1 [a’ +=+ b’, c’ +=+ d’]

        abcd’ :: Data [Word8]
        abcd’ = freezePush1 $ (thawPull1 ab’) +=+ (thawPull1 cd’)

        res ::  State
        res =   arrToPull (Z :. 4 :. 4) abcd’ -- reshape (Z :. 4 :. 4 ) abcd’
```

A.5 C, handwritten

Following the C99 standard, the following is a sample implementation of the AES main round for encrypting one block of data. The goal was to present a readable and straightforward general implementation, instead of a convoluted one that was optimized for a particular platform. As a more general optimization, however, loops were avoided.[10]

```
////////////////////////////////////////////////////////////////////////////////
// AES-128: Main Round (Encryption)
////////////////////////////////////////////////////////////////////////////////
#include <stdio.h>


const unsigned char sbox[256] =
{
  0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,
  // values omitted
  0x16
};

const unsigned char mul2[256] =
{
  0x00,0x02,0x04,0x06,0x08,0x0a,0x0c,0x0e,0x10,0x12,0x14,0x16,0x18,0x1a,0x1c,
  // values omitted
  0xe5
};

const unsigned char mul3[256] =
{
  0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,0x18,0x1b,0x1e,0x1d,0x14,0x17,0x12,
  // values omitted
  0x1a
};


////////////////////////////////////////////////////////////////////////////////
// SubBytes
////////////////////////////////////////////////////////////////////////////////

void SubBytes(unsigned char state[][4])
{
  state[0][0] = sbox[state[0][0]];
  state[0][1] = sbox[state[0][1]];
  state[0][2] = sbox[state[0][2]];
  state[0][3] = sbox[state[0][3]];
  state[1][0] = sbox[state[1][0]];
  state[1][1] = sbox[state[1][1]];
  state[1][2] = sbox[state[1][2]];
  state[1][3] = sbox[state[1][3]];
  state[2][0] = sbox[state[2][0]];
  state[2][1] = sbox[state[2][1]];
  state[2][2] = sbox[state[2][2]];
  state[2][3] = sbox[state[2][3]];
  state[3][0] = sbox[state[3][0]];
  state[3][1] = sbox[state[3][1]];
  state[3][2] = sbox[state[3][2]];
  state[3][3] = sbox[state[3][3]];
}
```

10 This code is a modification and simplification of the AES implementation developed by Brad Conte, available at https://github.com/B-Con/crypto-algorithms (accessed February 8, 2016). Relevant parts of it were rewritten to more closely resemble the Feldspar implementation on a syntactic level. Note that `main()` calls a helper function due to a limitation of the benchmarking tool Criterion. The main method is necessary for running this C program on its own, but has to be commented out when benchmarking.

```
////////////////////////////////////////////////////////////////////////////////
// ShiftRows
////////////////////////////////////////////////////////////////////////////////

void ShiftRows(unsigned char state[][4])
{
  int t;

  t           = state[1][0];
  state[1][0] = state[1][1];
  state[1][1] = state[1][2];
  state[1][2] = state[1][3];
  state[1][3] = t;

  t           = state[2][0];
  state[2][0] = state[2][2];
  state[2][2] = t;
  t           = state[2][1];
  state[2][1] = state[2][3];
  state[2][3] = t;

  t           = state[3][0];
  state[3][0] = state[3][3];
  state[3][3] = state[3][2];
  state[3][2] = state[3][1];
  state[3][1] = t;
}


////////////////////////////////////////////////////////////////////////////////
// MixColumns
////////////////////////////////////////////////////////////////////////////////

void MixColumns(unsigned char state[][4])
{
  unsigned char v0, v1, v2, v3; // temporary values

  v0           = state[0][0];
  v1           = state[1][0];
  v2           = state[2][0];
  v3           = state[3][0];
  state[0][0]  = mul2[v0] ^ mul3[v1] ^ v2       ^ v3;
  state[1][0]  = v0       ^ mul2[v1] ^ mul3[v2] ^ v3;
  state[2][0]  = v0       ^ v1       ^ mul2[v2] ^ mul3[v3];
  state[3][0]  = mul3[v0] ^ v1       ^ v2       ^ mul2[v3];

  v0           = state[0][1];
  v1           = state[1][1];
  v2           = state[2][1];
  v3           = state[3][1];
  state[0][1]  = mul2[v0] ^ mul3[v1] ^ v2       ^ v3;
  state[1][1]  = v0       ^ mul2[v1] ^ mul3[v2] ^ v3;
  state[2][1]  = v0       ^ v1       ^ mul2[v2] ^ mul3[v3];
  state[3][1]  = mul3[v0] ^ v1       ^ v2       ^ mul2[v3];

  v0           = state[0][2];
  v1           = state[1][2];
  v2           = state[2][2];
  v3           = state[3][2];
  state[0][2]  = mul2[v0] ^ mul3[v1] ^ v2       ^ v3;
  state[1][2]  = v0       ^ mul2[v1] ^ mul3[v2] ^ v3;
  state[2][2]  = v0       ^ v1       ^ mul2[v2] ^ mul3[v3];
  state[3][2]  = mul3[v0] ^ v1       ^ v2       ^ mul2[v3];

  v0           = state[0][3];
  v1           = state[1][3];
  v2           = state[2][3];
  v3           = state[3][3];
  state[0][3]  = mul2[v0] ^ mul3[v1] ^ v2       ^ v3;
  state[1][3]  = v0       ^ mul2[v1] ^ mul3[v2] ^ v3;
  state[2][3]  = v0       ^ v1       ^ mul2[v2] ^ mul3[v3];
  state[3][3]  = mul3[v0] ^ v1       ^ v2       ^ mul2[v3];
}


////////////////////////////////////////////////////////////////////////////////
// AddRoundKey
////////////////////////////////////////////////////////////////////////////////

void AddRoundKey(unsigned char state[][4]
                ,unsigned char key[]     )
{
   state[0][0] ^= key[0];
   state[0][1] ^= key[1];
   state[0][2] ^= key[2];
   state[0][3] ^= key[3];

   state[1][0] ^= key[4];
   state[1][1] ^= key[5];
   state[1][2] ^= key[6];
   state[1][3] ^= key[7];

   state[2][0] ^= key[8];
   state[2][1] ^= key[9];
   state[2][2] ^= key[10];
   state[2][3] ^= key[11];

   state[3][0] ^= key[12];
   state[3][1] ^= key[13];
   state[3][2] ^= key[14];
   state[3][3] ^= key[15];
}



////////////////////////////////////////////////////////////////////////////////
// Main
////////////////////////////////////////////////////////////////////////////////
void aes_mainround()
{
// arbitrary hardcoded values for benchmarking
  unsigned char in[16] = // plain text
  { 0x32, 0x88, 0x31, 0xe0,
    0x43, 0x5a, 0x31, 0x37,
    0xf6, 0x30, 0x98, 0x07,
    0xa8, 0x8d, 0xa2, 0x34 };

  unsigned char out[16] = // cipher text
  { 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00 };

  unsigned char key[16] =
  { 0x2b, 0x28, 0xab, 0x09,
    0x7e, 0xae, 0xf7, 0xcf,
    0x15, 0xd2, 0x15, 0x4f,
    0x16, 0xa6, 0x88, 0x3c };

    unsigned char state[4][4];
    state[0][0] = in[0];
    state[0][1] = in[1];
    state[0][2] = in[2];
    state[0][3] = in[3];
    state[1][0] = in[4];
    state[1][1] = in[5];
    state[1][2] = in[6];
    state[1][3] = in[7];
    state[2][0] = in[8];
    state[2][1] = in[9];
    state[2][2] = in[10];
    state[2][3] = in[11];
    state[3][0] = in[12];
    state[3][1] = in[13];
    state[3][2] = in[14];
    state[3][3] = in[15];

    SubBytes(state);
    ShiftRows(state);
    MixColumns(state);
    AddRoundKey(state, key);

    out[0]  = state[0][0];
    out[1]  = state[0][1];
    out[2]  = state[0][2];
    out[3]  = state[0][3];
    out[4]  = state[1][0];
    out[5]  = state[1][1];
    out[6]  = state[1][2];
    out[7]  = state[1][3];
    out[8]  = state[2][0];
    out[9]  = state[2][1];
    out[10] = state[2][2];
    out[11] = state[2][3];
    out[12] = state[3][0];
    out[13] = state[3][1];
    out[14] = state[3][2];
    out[15] = state[3][3];

  /*
  for (unsigned int i = 0; i < 16; i++)
    printf("%02d ",out[i]);
  puts("");
  */

}

int main()
{
  aes_mainround();
  return 0;
}
```

A.6 C, generated by Feldspar

This subsection presents C code as it is generated by Feldspar. Note that we occasionally added blank lines and line breaks in order to increase readability. Furthermore, there are occasional comments, offset by `//`, to highlight how this code corresponds to the Feldspar source code given previously.

We start with `subBytes`:

```
void test(struct s_2_arr_unsignedS32_arr_unsignedS8 * v0,
          struct s_2_arr_unsignedS32_arr_unsignedS8 * out)
{
  uint32_t v5;
  uint32_t v6;
  struct array * v7 = NULL;
  uint32_t len8;

  v5 = at(uint32_t,(*v0).member1,0);
  v6 = at(uint32_t,(*v0).member1,1);
  v7 = initArray(v7, sizeof(uint8_t), 256);  // sbox
  at(uint8_t,v7,0) = 99;
  // 252 lines of assignments omitted
  at(uint8_t,v7,255) = 22;

  (*out).member1 = setLength((*out).member1, sizeof(uint32_t), 2);
  at(uint32_t,(*out).member1,0) = v5;
  at(uint32_t,(*out).member1,1) = v6;
  len8 = (v6 * v5);

  (*out).member2 = initArray((*out).member2, sizeof(uint8_t), len8);
  for (uint32_t v4 = 0; v4 < len8; v4 += 1)
  {
    at(uint8_t,(*out).member2,v4) = at(
      uint8_t,v7,((uint32_t)(at(
        uint8_t,(*v0).member2,(((v4 / v5) * v5) + (v4 % v5))))));
  }
  freeArray(v7);
}
```

This is `shiftRows`:

```
void test(struct s_2_arr_unsignedS32_arr_unsignedS8 * v0,
          struct s_2_arr_unsignedS32_arr_unsignedS8 * out)
{
  uint32_t v5;
  uint32_t v6;
  uint32_t len8;
  uint32_t v7;

  v5 = at(uint32_t,(*v0).member1,0);
  v6 = at(uint32_t,(*v0).member1,1);

  (*out).member1 = setLength((*out).member1, sizeof(uint32_t), 2);
  at(uint32_t,(*out).member1,0) = v5;
  at(uint32_t,(*out).member1,1) = v6;

  len8 = (v6 * v5);
  (*out).member2 = initArray((*out).member2, sizeof(uint8_t), len8);

  for (uint32_t v4 = 0; v4 < len8; v4 += 1)
  {
    v7 = (v4 / v5);
    at(uint8_t,(*out).member2,v4) = at(
      uint8_t,(*v0).member2,((v7 * v5) + (((v4 % v5) + v7) % 4)));
  }
}
```

The function `mixColumns` was implemented in two ways. The first version is based on extracting columns from the provided `State` before applying `MixColumn`. The alternative definition of `mixColumns` generated a less C code, as it used available abstractions in the language. Reshaping, for instance, does not change the values of an array but merely changes the dimensions, as they are specified in a tuple associated to it. The generated code overlaps largely, due to sharing many operations.

```
void test(struct s_2_arr_unsignedS32_arr_unsignedS8 * v0,
          struct s_2_arr_unsignedS32_arr_unsignedS8 * out)
{
  uint32_t v12432;
  struct array * v12417 = NULL;
  // variable declarations; most code omitted

  v12431 = at(uint32_t,(*v0).member1,0);
  v12432 = (v12431 << 1);
  v12433 = (v12431 * 3);
  v12413 = at(uint8_t,(*v0).member2,0);
  v12414 = at(uint8_t,(*v0).member2,v12431);
  v12415 = at(uint8_t,(*v0).member2,v12432);
  v12416 = at(uint8_t,(*v0).member2,v12433);

  v12417 = initArray(v12417, sizeof(uint8_t), 256);  // cf. mult2
  at(uint8_t,v12417,0) = 0;
  at(uint8_t,v12417,1) = 2;
  // code omitted

  v12418 = initArray(v12418, sizeof(uint8_t), 256);  // cf. mult3
  at(uint8_t,v12418,0) = 0;
  at(uint8_t,v12418,1) = 3;
  // code omitted

  v12419 = at(uint8_t,(*v0).member2,1);
  v12420 = at(uint8_t,(*v0).member2,(v12431 + 1));
  v12421 = at(uint8_t,(*v0).member2,(v12432 + 1));
  v12422 = at(uint8_t,(*v0).member2,(v12433 + 1));
  // code omitted

  at(uint8_t,v12314,0) = v12413;
  at(uint8_t,v12314,1) = v12414;
  at(uint8_t,v12314,2) = v12415;
  at(uint8_t,v12314,3) = v12416;

  // XOR operations:
  v12315 = (((at(uint8_t,v12417,((uint32_t)(at(uint8_t,v12314,0))))
    ^ at(uint8_t,v12418,((uint32_t)(at(uint8_t,v12314,1)))))
    ^ at(uint8_t,v12314,2))
    ^ at(uint8_t,v12314,3));

  v12320 = setLength(v12320, sizeof(uint8_t), 4);
  at(uint8_t,v12320,0) = v12413;
  at(uint8_t,v12320,1) = v12414;
  at(uint8_t,v12320,2) = v12415;
  at(uint8_t,v12320,3) = v12416;
  // code omitted

  v12360 = setLength(v12360, sizeof(uint8_t), 8);
  for (uint32_t v5388 = 0; v5388 < 4; v5388 += 1)
  {
    at(uint8_t,v12360,v5388) = at(uint8_t,v12334,v5388);
    at(uint8_t,v12360,(v5388 + 4)) = at(uint8_t,v12359,v5388);
  }

  // code omitted
  v12411 = setLength(v12411, sizeof(uint8_t), 8);
  for (uint32_t v10514 = 0; v10514 < 4; v10514 += 1)
  {
    at(uint8_t,v12411,v10514) = at(uint8_t,v12385,v10514);
    at(uint8_t,v12411,(v10514 + 4)) = at(uint8_t,v12410,v10514);
  }
  v12412 = setLength(v12412, sizeof(uint8_t), 16);
  for (uint32_t v3081 = 0; v3081 < 8; v3081 += 1)
  {
    at(uint8_t,v12412,v3081) = at(uint8_t,v12360,v3081);
    at(uint8_t,v12412,(v3081 + 8)) = at(uint8_t,v12411,v3081);
  }
  (*out).member1 = initArray((*out).member1, sizeof(uint32_t), 2);
  at(uint32_t,(*out).member1,0) = 4;
  at(uint32_t,(*out).member1,1) = 4;
  (*out).member2 = initArray((*out).member2, sizeof(uint8_t), 16);
  for (uint32_t v4 = 0; v4 < 16; v4 += 1)
  {
    at(uint8_t,(*out).member2,v4) = at(uint8_t,v12412,(((v4 % 4) << 2) + (v4 / 4)));
  }

  freeArray(v12417);
  // code omitted
}
```

Lastly, here is the code generated for `addRoundKey`:

```
void test(struct s_2_arr_unsignedS32_arr_unsignedS8 * v0,
         struct s_2_arr_unsignedS32_arr_unsignedS8 * v1,
         struct s_2_arr_unsignedS32_arr_unsignedS8 * out)
{
  uint32_t v12;
  uint32_t v13;
  uint32_t v6;
  uint32_t v7;
  uint32_t len14;
  uint32_t v10;
  uint32_t v11;

  v12 = at(uint32_t,(*v0).member1,0);
  v13 = at(uint32_t,(*v1).member1,0);
  v6 = min(v12, v13);
  v7 = min(at(uint32_t,(*v0).member1,1), at(uint32_t,(*v1).member1,1));

  (*out).member1 = setLength((*out).member1, sizeof(uint32_t), 2);
  at(uint32_t,(*out).member1,0) = v6;
  at(uint32_t,(*out).member1,1) = v7;

  len14 = (v7 * v6);
  (*out).member2 = initArray((*out).member2, sizeof(uint8_t), len14);
  for (uint32_t v5 = 0; v5 < len14; v5 += 1)
  {
    v10 = (v5 / v6);
    v11 = (v5 % v6);
    at(uint8_t,(*out).member2,v5) = (at(
      uint8_t,(*v0).member2,((v10 * v12) + v11)) ^ at(
        uint8_t,(*v1).member2,((v10 * v13) + v11)));
  }
}
```