Verified Post-Quantum Cryptography

Verifying The McEliece Cryptosystem With Refinement Types

Tom Arnold
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
tca4384@rit.edu

Abstract

New cryptosystems are being developed and standardized to resist attacks from quantum computers; one such cryptosystem is the McEliece cryptosystem which owes its security to coding theory and binary Goppa codes. Implementation of the McEliece cryptosystem involves linear algebra, coding theory, and arithmetic with polynomials of binary Galois field elements.

Cryptosystems are one class of software where the absence of defects is highly desirable; this is where formal verification is useful. Formal verification can help prove mathematically the correct implementation of these systems.

We present a study of formal verification applied to the domain of post-quantum cryptography by implementing the McEliece cryptosystem in LiquidHaskell and verifying it using refinement types. This includes all of the supporting mathematics required for the cryptosystem including linear algebra, coding theory, polynomial arithmetic, and binary Galois field arithmetic. As part of this study we show that refinement types are a useful tool for reducing the chance of security breaking bugs during implementation of a cryptosystem.

I. Introduction

Widely used public key cryptosystems are vulnerable to attacks from quantum computers. These cryptosystems are based on one-way functions for which there is no efficient solution to calculate the inverse of the function; the security of the system is built on this fact. Quantum computing has introduced new algorithms which can efficiently compute solutions to some of these problems. As a result, cryptographers are working to implement new cryptosystems based on problems that are not efficiently solvable by quantum or digital computers [1].

The McEliece cryptosystem is one such system that dates back to the 70s when it was proposed by Robert McEliece. The system is based on coding theory and to break it an attacker would have to solve the general decoding problem which is NP-complete [6]. A modern variant of McEliece called Classic McEliece is one of the finalists in the NIST Post-Quantum standardization effort [3].

Formal verification can be used to prove the absence of bugs in a way that automated or manual testing cannot. Typically such verification is only done for software which must be held to a high standard. Cryptography is one such field where even simple errors can have severe real-world consequences [4].

In the following paper we show how a post-quantum cryptosystem (McEliece) can be implemented and formally verified using refinement types with LiquidHaskell [5]. We also examine the effort involved in performing such verification so as to better understand the costs and benefits involved.

II. Post-Quantum Cryptography

Post-quantum cryptography is a new field of cryptography focused on the creation of cryptosystems which are secure despite the existence of quantum computers. Quantum computers are able to natively execute quantum algorithms which can solve some problems much more efficiently than a classical computer could. This is a problem for cryptography because public-key cryptography is built on one-way functions, i.e. functions that are easy to compute in one direction but difficult to compute the reverse of, and for some problems there exist quantum algorithms that break this assumption. One example of this concern is Shor's algorithm and the integer factoring problem which is used by the famous RSA public-key cryptosystem.

Although large-scale quantum computers do not yet exist, it seems likely that they will in the near future and replacing our existing pre-quantum cryptosystems will take years. Additionally anything encrypted today with pre-quantum cryptography could be broken once someone is able to develop a large-scale quantum computer. As a result of these concerns, NIST is currently running a post-quantum cryptography competition with the goal of standardizing several post-quantum cryptosystems so that government and industry can begin requiring and using them.

One of the cryptosystems being standardized by NIST is Classic McEliece [12], a key-encapsulation mechanism (KEM) built on the Niederreiter public-key cryptosystem which is a variant of the original McEliece public-key cryptosystem proposed by Robert McEliece in the 70s.

While the McEliece and Niederreiter schemes are quite old, they are based on a problem (i.e. general decoding) which does not have a quantum algorithm and is NP-complete, therefore they are post-quantum safe. These systems have not gotten much attention from cryptographers until recently

due to their large key sizes. Both systems use large matrices of bits for their private and public keys, and while they work with any linear code, binary Goppa codes are the only ones that have proven to be secure. The primary difference between the two systems is that Niederreiter uses a parity check matrix instead of a generator matrix as its public key and as a result the ciphertext is a syndrome vector instead of a garbled codeword. An advantage of the Niederreiter system is that the parity matrix can be represented in standard form $H = (I|M)$ which allows the key size to be halved. The Niederreiter system can also be used for digital signatures unlike the original McEliece system [9].

Because of their similarity with each other and given that a modern post-quantum cryptosystem (Classic McEliece) is based on one of them, the McEliece and Niederreiter schemes are an accessible starting point for formal verification; they are relatively simple but also relevant.

III. Verification Techniques

Formal verification can be used to prove the absence of certain types of bugs in a program. This technique typically uses some kind of theorem prover to mathematically prove certain properties about a program. One way to apply this technique is to integrate it into the type system of a programming language in the form of refinement types.

Refinement types are a way to refine a type to consist of more specific values. For example, suppose we have a function f which accepts an integer parameter x and performs a runtime assertion to check that the parameter is even. Instead of doing this check at runtime we could refine the type of x from integer to even integer, i.e. integers divisible by two or $x : \{Int | x \bmod 2 = 0\}$ which reads as "x is an integer such that x is divisible by 2". More complicated refinements are possible, including refining the type of a parameter relative to another parameter, or refining the return type of a function based on its inputs [13].

Haskell is a functional programming language with an advanced type system. Haskell is typically used as a research language for programming language theory. Code written in Haskell is strongly-typed and the language provides many features to make it convenient to work with this system; for example Haskell allows developers to implement custom number types by defining function to convert to/from integers and to perform arithmetic. A plugin to Haskell called LiquidHaskell provides refinement types in the form of comment annotations that the plugin parses during compilation. After parsing the code, LiquidHaskell generates constraints and invokes the Z3 theorem prover to determine if the constraints are satisfiable; if they are not then this usually indicates a flaw in the program logic and an error message is returned to the developer. It is important to note that the LiquidHaskell annotations are just annotations and therefore only affect code being compiled with the LiquidHaskell plugin; they do not affect the generated code or the runtime of the program at all, nor are they checked in interpreted mode when using the $GHCi$ REPL (read-eval-print loop) [5].

In the next sections we will discuss in detail what refinement types can be used for.

A. Size-Aware API

When working with vectors and matrices one often must be aware of their dimensions so as to avoid accessing an element outside of the bounds of the data structure. Typically these bounds checks are performed at runtime but refinement types allow them to be lifted into compile time checks. This works by tracking the size of the data structure at the type level.

Tracking the size of the data at the type level has several requirements. First we must define a measure which is a function that can be converted into first-order logic and can be used to determine a size. For a list we use a simple recursive function `size` which counts the elements in the list. We then annotate this with a LiquidHaskell annotation `{-@ measure size @-}` stating that the function `size` is a measure. Now we use another LiquidHaskell annotation `{-@ type ListN a N = {v: List a | size v = N} @-}` to define a type alias for a list of size N; this reads as "List of some type such that the size of the list is N."

This technique provides us with several tools and responsibilities. First, it allows us to write functions that take a list and a index and verify (at compile time) that the index is in-bounds simply by writing a refinement that states that the index is a natural number that is less than the size of the list. It also allows us to write functions that take multiple lists of the same size, e.g. to zip two lists, and the size check is performed at compile time. This can be extended to other data types as well, for example if we represent a matrix as a list of lists then this gives us a way to check at compile time that two matrices have compatible dimensions when adding or multiplying them [5].

The burden of this technique is that all code dealing with these lists must be "size-aware", i.e. a function that transforms a list of size N must be annotated to return a list of size N, and any code within the function must be verifiable by LiquidHaskell to preserve the size of the list.

B. Termination Checking

When writing recursive (or looping) code there is a chance that the program will loop endlessly if the algorithm is implemented incorrectly or if the parameters are invalid. Refinement types can be used to verify that a recursive function always terminates. To do this one uses a termination measure, which similar to a measure used to represent the size of a data structure can be used to show that a parameter or combination of parameters is decreasing towards the base case of the recursion. This can be used by LiquidHaskell to perform a proof-by-induction that the function terminates.

C. Derived Constraints

An useful property of refinement types is that their effects permeate the code-base. LiquidHaskell can infer refinements

similar to how Haskell infers types. This means that the programmer typically only has to annotate pre and post conditions on functions and data types and LiquidHaskell is able to "fill in" the details in between. The takeaway is that refinement types are easy to use and are not overly verbose.

IV. Implementation

Having discussed refinement types, we will now dive into implementing and verifying the McEliece and Niederreiter post-quantum cryptosystems in Haskell by presenting select parts of the implementation. The full source-code is available at https://bitbucket.org/Tom9729/csci788/src/master/ and can be compiled using the Haskell Tool Stack at https://www.haskellstack.org/.

A. Support Code

Many libraries exist to perform matrix and Galois field math in Haskell, but none of them are verified with refinement types. While LiquidHaskell provides a mechanism to introduce refinements to an external library, for this project we chose to simply re-implement the required math for the McEliece and Niederreiter schemes.

1) Matrices: The McEliece and Niederreiter schemes use matrices and vectors for their keys, plaintext, and ciphertexts. In this project we implemented a simple linear algebra package consisting of matrices, vectors, and several useful operations, and then verified it using refinement types.

First we present the data structure for our matrices in Figure 1. We represent matrices as a count of rows and columns, and a vector of vectors. We use refinements to say that the rows and columns must be positive non-zero integers (Pos being a type alias for `v:{Nat | v > 0}`, it does not make sense to have a matrix with zero rows/columns) and that the number of elements in the row and column vectors match the number of rows and columns in the matrix. We also define a type alias $MatrixN$ for a matrix with R rows and C columns.

```
{-@ data Matrix a = M { mRow :: Pos
                      , mCol :: Pos
                      , mElts :: VectorN (VectorN a mCol)
                                         mRow } @-}
{-@ predicate Dims M R C = mRow M = R && mCol M = C @-}
{-@ type MatrixN a R C = {v:Matrix a | Dims v R C} @-}
```

Fig. 1. Matrix type.

Armed with our refined matrix type we now proceed to do something useful. In Figure 2 we present a function that appends the columns from one matrix to another. Because this operation only makes sense for matrices with the same number of rows we start with a refinement on the second parameter stating that it has the same number of rows as the first parameter. This ensures that if the function is called with matrices with different numbers of rows then it will be a compile error. This is a pre-condition refinement.

To ensure that the output from the function is of a known size we create a post-condition refinement stating that the number of rows of the output match that of the first input (given that both inputs have the same number of rows it would be equally true to match the second input), and that the number of columns of the output equals the sum of the number of columns from the inputs, e.g. if we append a $3x2$ and a $3x1$ matrix we would get a $3x3$ matrix.

```
{-@ matAppend :: m:Matrix a
              -> n:{Matrix a | mRow m = mRow n}
              -> o:{Matrix a | mRow o = mRow m && mCol o ↩
                               = mCol m + mCol n} @-}
matAppend m@(M mr mc mrows) n@(M _ nc nrows) =
  M mr (mc + nc) $ vZipWith (\mrow nrow ->
                               vConcat mrow nrow) mrows ↩
                                    nrows
```

Fig. 2. Appending two matrices.

2) Polynomials: The McEliece and Niederreiter schemes use polynomials during the key generation and decryption processes. In Figure 3 we present a type for a generic univariate (single-variable) polynomial represented as a vector of coefficients and a natural number as the degree of the polynomial. We setup a simple refinement stating that there are one more coefficients than the degree, so a degree zero polynomial has one coefficient. Note that a empty coefficient list is impossible because a negative degree polynomial is unrepresentable.

```
{-@ data Poly a = P { pDeg :: Nat
                    , pCoefs :: VectorN a {pDeg + 1}
                    } @-}
{-@ type PolyN a N = {p:Poly a | pDeg p = N} @-}
```

Fig. 3. Polynomials as vectors of coefficients.

In Figure 4 we present a function that accesses one of the coefficients from a polynomial in a typesafe manner. The function takes a polynomial p and index i and has a refinement that requires the index to be positive and less than the size of the coefficients vector.

```
{-@ pCoef :: (Num a) => p:Poly a
          -> i:{Nat | i <= pDeg p} -> a @-}
pCoef :: (Num a) => Poly a -> Int -> a
pCoef p@(P d ps) i = vGet ps i
```

Fig. 4. Accessing a polynomial coefficient safely.

3) Binary Galois Fields: Key generation and decryption for the McEliece and Niederreiter schemes uses polynomials of binary Galois field elements; i.e. the coefficients of the polynomials are essentially polynomials themselves. Thus we implemented binary Galois fields as a special case of polynomial, extending functionality as necessary.

Fields are represented by a characteristic polynomial, and only binary fields are supported. The field polynomial is a polynomial of bits and field elements consist of a reference to the field polynomial as well as another polynomial of bits representing the element.

Addition and subtraction are implemented as a bitwise XOR. Multiplication is inherited from the polynomial package except with an extra step to reduce by the field polynomial to keep the result valid. Division is implemented as multiplication by the inverse, found by performing EEA with the field polynomial.

```
type GF2Poly = Poly Bit

{-@ data GF2Element =
          E { eField :: f:{GF2Poly | f /= 0}
            , ePoly :: GF2Poly
            } @-}
```

Fig. 5. Binary Galois fields as polynomials.

Figure 6 shows our implementation of binary Galois field reduction, a process which reduces field elements by reducing them by the field polynomial until they are of a lesser degree. At each iteration we pad the field polynomial to the right with zeroes until it is of the same degree as the field element, then we add them which performs an XOR operation on their binary coefficients. Because a polynomial of non-zero degree must have a non-zero leading coefficient, this procedure must result in a smaller polynomial, and thus the first parameter to our function is decreasing and we have proved termination.

Here we are confronted with our first failing in the pursuit of well-verified code. We must show that the degree of a polynomial of bits decreases when added to another polynomial of bits of the same degree, and to do this we must create an invariant that states that the leading coefficient of a non-zero degree polynomial is non-zero. Unfortunately we were unable to determine how to do this. The reader should note that this was likely more of a limitation of our understanding than one of LiquidHaskell.

```
{-@ reduceGF2 :: p:Poly a -> f:Poly a -> Poly a @-}
{-@ lazy reduceGF2 @-}
reduceGF2 p f@(P d v) =
  let delta = pDeg p - pDeg f
  in
    if delta < 0 || p == 0 || f == 1
    then p
    else
      let fd' = pDeg p
          f' = P fd' $ vExtendRight v $ fd' + 1
      in reduceGF2 (p + f') f
```

Fig. 6. Binary Galois field reduction.

4) Haskell Typeclasses: Haskell has a feature called typeclasses which allows code to be written using generic types; this is similar to the concept of interfaces in Java [7]. A typeclass defines a set of functions that can be called on implementations of that type, and Haskell defines many standard typeclasses. Of interest to us are two: Num and $Integral$ which define basic arithmetic and whole-number division respectively [8]. For this project we implemented these typeclasses for polynomials (Figure 7) and Galois fields, allowing us to write generic implementations of algorithms such as the Extended Euclid Algorithm that operated on both integers and polynomials of field elements. This demonstrates one appeal of using Haskell, which is writing terse and reusable code.

One of the challenges with implementing typeclasses for something like a polynomial or Galois field element is how to handle all of the methods in the class. For example, to implement Num we are required to implement absolute value, but what does that mean for a polynomial? For Galois field elements if we implicitly convert an integer to a field element how do we know what field that element should belong to? While an incomplete implementation of a class is possible it does leave strange edge cases that for now we were unable to handle.

```
instance (Eq a, Num a) => (Num (Poly a)) where
  (+) p q = addPoly p q
  (-) p q = subPoly p q
  (*) p q = multPoly p q
  fromInteger x = P 0 $ V 1 [fromInteger x]

instance (Integral a) => Integral (Poly a) where
  quotRem a b = quoRemPoly a b
```

Fig. 7. Implementing Num and Integral typeclasses for polynomials.

5) Extended Euclidean Algorithm (EEA): The Extended Euclid Algorithm (also known as EEA) computes the greatest common divisor d of two numbers and returns it as well as the coefficients xy of Bezout's identity such that $d = a \times x + b \times y$. This is typically used in cryptography to find the multiplicative inverse of a Galois field element.

Implementation of the EEA in Haskell is a straightforward conversion from textbook pseudocode iteration to recursion [11]. Of note is that because of Haskell's type system we are able to implement the EEA once for numbers, polynomials, and field elements. Here refinement types provide us with two additional checks beyond what the Haskell type system provides.

First we refine the second parameter to be non-zero because we perform a division by it and that would result in a division-by-zero runtime error. Second we refine the inner recursive loop go to have a termination condition. As long as $b0 > r$ the remainder of $b0/r$ is decreasing, and since $r = 0$ is our base case we have shown inductively that the function will always terminate.

```
{-@ extEuclidAlg :: (Integral a)
                 => a:a
                 -> b:{a | b /= 0}
                 -> (a, a, a) @-}
extEuclidAlg a b =
  let (q, r) = a `quotRem` b
  in
    go 0 1 1 0 a b q r
  where
    {-@ go :: (Integral a)
           => t0:a -> t:a -> s0:a -> s:a -> a0:a -> b0:a ↩
              -> q:a
           -> r:{a | b0 > r}
           -> (a,a,a) / [r] @-}
    go t0 t s0 s a0 b0 q r =
      if r == 0 then (b0, s, t)
      else
        let t' = t0 - q * t
            s' = s0 - q * s
            (q',r') = b0 `quotRem` r
        in
          go t t' s s' b0 r q' r'
```

Fig. 8. Extended Euclid Algorithm.

B. Patterson's Algorithm

Patterson's algorithm is used to decode Goppa encoded messages with up to t errors. The high-level implementation of this is somewhat straightforward but it consists of several helper functions, and the mathematics are best explained by other papers (see [1]).

First we implemented a modified Extended Euclid Algorithm as $goppaExtEuclidAlg$ [10]. This version operates on polynomials and returns when the degree of the polynomials reaches a certain point. Unlike our straightforward implementation of the EEA this one happens to not require any explicit refinement annotations for LiquidHaskell to verify that it terminates.

```
goppaExtEuclidAlg a b = go b a 0 1 1 0
  where
    deg = pDeg a
    go 0 lastr _ lastu _ lastv = (lastr, lastu, lastv)
    go r lastr u lastu v lastv =
      let (lastr', (q, r')) = (r, lastr `quotRem` r)
          (u', lastu') = (lastu - q * u, u)
          (v', lastv') = (lastv - q * v, v)
      in
        if pDeg u' <= (deg - 1) `quot` 2 && pDeg r' <= ↩
            deg `quot` 2
        then (lastu', lastr', lastv')
        else go r' lastr' u' lastu' v' lastv'
```

Fig. 9. Modified Extended Euclid Algorithm.

Next we implemented a procedure $goppaSplit$ to split a polynomial into even and odd coefficients as required for Patterson's algorithm. One thing to note is that if we split a zero degree polynomial the function still satisfies the invariant that a polynomial must have one coefficient by adding a zero coefficient.

```
goppaSplit (P d (V n xs)) = (p0, p1)
  where
    p0 = P (nEvens - 1) $ V nEvens evens'
    p1 = P (nOdds - 1) $ V nOdds odds'

    {-@ evens' :: ListGE GF2Element 1 @-}
    (evens, odds) = altList $ reverse' $ map' sqrtGF2 xs
    evens' = if size evens == 0 then [0] else evens
    odds' = if size odds == 0 then [0] else odds

    nEvens = size evens'
    nOdds = size odds'
```

Fig. 10. Splitting a polynomial into even/odd coefficients.

Next we implement a procedure $goppaInverse$ to find the inverse of a polynomial modulo the Goppa polynomial.

```
{-@ goppaInverse :: p:Poly GF2Element
                 -> g:{Poly GF2Element | g /= 0}
                 -> Poly GF2Element
  @-}
goppaInverse p 0 = 0
goppaInverse p g = u `mod` g
  where
    (d, u, v) = extEuclidAlg p g
```

Fig. 11. Polynomial inverse modulo the Goppa polynomial.

Now we have everything we need to implement Patterson's algorithm. This is used during McEliece and Niederreiter decryption to find the error locator polynomial which is used to decode the ciphertext. The only refinement we need to specify is that the Goppa polynomial must be non-zero.

```
{-@ pattersonsAlgorithm :: x:_
                        -> g:{_ | g /= 0}
                        -> s:_ -> _ @-}
pattersonsAlgorithm x g syndrome = u^2 + x * v^2
  where
    (g0, g1) = goppaSplit g
    w = g0 * goppaInverse g1 g
    t = polyInvMod syndrome g
    (t0, t1) = goppaSplit $ t + x
    r = t0 + w * t1
    (_, u, v) = goppaExtEuclidAlg g r
```

Fig. 12. Patterson's algorithm.

C. McEliece Cryptosystem

We now present a toy example of the McEliece scheme using Goppa codes.

1) Key Setup: First we must define some private parameters used during key generation and decryption. f is the field polynomial, g is the Goppa polynomial, z and x are primitive elements used to generate the code.

$codeLocators$ is used to generate the private key matrix and $syndLocators$ (referred to by Roering as $SyndromeCalculator$) is used during the decryption process to convert the ciphertext to a polynomial.

```
-- (z^3 + z + 1)
f = P 3 $ V 4 [1, 0, 1, 1]

-- X^2 + X + (1)
g = P 2 $ V 3 [E f $ P 0 $ V 1 [1], E f $ P 0 $ V 1 [1], ↩
     E f $ P 0 $ V 1 [1]]

-- (z)
z = E f $ P 1 $ V 2 [1, 0]

-- X
x = P 1 $ V 2 [E f $ P 0 $ V 1 [1], 0]

-- [() ,(1) ,(z) ,(z^2) ,(z + 1) ,(z^2 + z) ,(z^2 + z + 1) ,(z^2↩
      + 1)]
codeLocators = (z*0):(1+z*0):(map' (\i -> z^i) $ nseq 6)

-- | X + (1) X (z^2)*X + (z^2 + z + 1) (z^2 + z)*X + (z +↩
     1) (z^2)*X + (z + 1) (z)*X + (z^2 + 1) (z)*X + (z^2 ↩
     + z + 1) (z^2 + z)*X + (z^2 + 1) |
syndLocators = V 8 $ [polyInvMod (x - (p c)) g | c <- ↩
     codeLocators]
  where
    p c = P 0 $ V 1 [c]
```

Fig. 13. McEliece private parameters.

We now present the private and public keys for this example. These were generated by SAGE [14] code from the above parameters (see [1] and [2]). The private key consists of *generator* derived from the code locators, *scrambler* a random matrix, and *permutation* a randomly shuffled identity matrix. Note that decryption uses the inverses of these matrices which is computed by the function $bitMatInv$ which performs elementary row operations to find the inverse of the matrix; also note that because the generator matrix is non-square we use the right-inverse. The public key $generator'$ consists of the product of the private key matrices.

```
generator = M 2 8 $ V 2
  [
    V 8 [1, 1, 0, 0, 1, 0, 1, 1]
  , V 8 [0, 0, 1, 1, 1, 1, 1, 1]
  ]

scrambler = M 2 2 $ V 2
  [
    V 2 [0, 1]
  , V 2 [1, 0]
  ]

permutation = M 8 8 $ V 8
  [
    V 8 [0, 1, 0, 0, 0, 0, 0, 0]
  , V 8 [0, 0, 0, 1, 0, 0, 0, 0]
  , V 8 [0, 0, 0, 0, 0, 0, 1, 0]
  , V 8 [1, 0, 0, 0, 0, 0, 0, 0]
  , V 8 [0, 0, 1, 0, 0, 0, 0, 0]
  , V 8 [0, 0, 0, 0, 0, 1, 0, 0]
  , V 8 [0, 0, 0, 0, 1, 0, 0, 0]
  , V 8 [0, 0, 0, 0, 0, 0, 0, 1]
  ]

-- | 1 0 1 0 1 1 1 1 |
-- | 0 1 1 1 1 0 0 1 |
generator' = (scrambler `matProduct` generator) `↩
     matProduct` permutation
```

Fig. 14. McEliece public and private keys.

2) Encryption: The encryption process for the McEliece scheme is fairly straightforward. The plaintext message is encoded as a bit string and a random error vector of weight 2 (i.e. Goppa polynomial degree) is added to it. Note that this random error is what makes the McEliece scheme probabilistic, a property which is not shared with the Niederreiter scheme. The result is multiplied by the public key matrix to obtain the ciphertext.

```
ptext = V 2 [1, 0]
err = V 8 [0, 0, 0, 0, 1, 0, 1, 0]

-- | 1 0 1 0 0 1 0 1 |
ctext = (ptextM `matProduct` generator') `matAdd` errM
  where
    ptextM = matFromVec ptext
    errM = matFromVec err
```

Fig. 15. McEliece encryption.

3) Decryption: Decryption for the McEliece scheme is a little more involved than encryption. First we use the inverse of the permutation matrix to obtain a modified ciphertext, then we use the syndrome locators to create the syndrome polynomial. Once we have that we can use Patterson's algorithm to find the error locator polynomial. By evaluating the code locators in the error locator polynomial we can find and remove the error from the ciphertext. Next we use the right-inverse of the generator matrix to remove the parity bits (transforming from codeword back to message) and finally we use the inverse of the scrambler matrix to retrieve the original plaintext message.

```
-- | 0 0 0 1 1 1 0 1 |
ctext' = ctext `matProduct` permutationInv

-- (z^2 + z)*X
syndrome = dotProduct syndLocators ctext''
  where
    ctext'' = for (matToVec ctext') (\x -> P 0 $ V 1 [E f↩
          $ P 0 $ V 1 [x]])

-- (z)*X^2 + X + (1)
errorLocator = pattersonsAlgorithm x g syndrome

-- | 0 0 1 0 0 0 1 0 |
ctextError = V 8 $ map' (\c -> if 0 == evalPoly ↩
     errorLocator c then 1 else 0)
          codeLocators

-- | 0 0 1 1 1 1 1 1 |
ctext''' = ctext' `matAdd` (matFromVec ctextError)

-- | 0 1 |
ctext'''' = ctext''' `matProduct` generatorInv

-- | 1 0 |
ptext' = ctext'''' `matProduct` scramblerInv
```

Fig. 16. McEliece decryption.

D. Niederreiter Cryptosystem

Next we look at a toy example of the Niederreiter scheme using Goppa codes. This is similar to the McEliece scheme

except that we use the parity check matrix instead of the generator matrix which means that our ciphertext is actually the syndrome instead of a codeword with random error. This property is what makes Niederreiter deterministic and thus suitable for use as a digital signature scheme (although we do not implement that here).

1) Key Setup: Similar to the McEliece scheme we first start with our key parameters. Note that in this example the syndrome locators are not generated; during the decryption process we use a different technique to compute the syndrome polynomial from the ciphertext.

```
-- (z^3 + z^2 + 1)
f = P 3 $ V 4 [1, 1, 0, 1]

-- X^2 + X + (1)
g = P 2 $ V 3 [E f $ P 0 $ V 1 [1], E f $ P 0 $ V 1 [1], ↩
     E f $ P 0 $ V 1 [1]]

-- (z)
z = E f $ P 1 $ V 2 [1, 0]

-- X
x = P 1 $ V 2 [E f $ P 0 $ V 1 [1], 0]

-- [() ,(1) ,(z^2) ,(z^2 + z + 1) ,(z^2 + z) ,(z) ,(z^2 + 1) ,(z↩
      + 1)]
codeLocators :: List GF2Element
codeLocators = (p*0):(1+p*0):(map' (\i -> p^i) $ nseq 6)
  where
    p = z^2
```

Fig. 17. Niederreiter private parameters.

Next we present our private and public key matrices in Figure 18. Like the McEliece example these were generated by SAGE code from the above parameters. Note that here we use the parity check matrix *parity* in place of the generator matrix.

2) Encryption: Niederreiter encryption is slightly simpler looking than McEliece encryption. Notice that there is no error vector added to the plaintext; in the Niederreiter scheme the plaintext is the error. This means that although (in this example) the plaintext string is of length 8 it can only have weight 2 (i.e. the degree of the Goppa polynomial).

```
ptext = V 8 [0, 0, 1, 0, 0, 0, 1, 0]

-- | 1 1 1 0 1 1 |
ctext = transpose $ parity' `matProduct` (transpose $ ↩
    matFromVec ptext)
```

Fig. 19. Niederreiter encryption.

3) Decryption: Decryption in the Niederreiter scheme is similar to the McEliece scheme, but note that we use the inverses of the scrambler and permutation matrices in a different order here. Also there is an implicit process required to decode the message from the plaintext once decryption is complete.

```
parity = M 6 8 $ V 6
  [
    V 8 [1, 0, 0, 1, 0, 0, 0, 1]
  , V 8 [0, 0, 0, 1, 0, 1, 1, 1]
  , V 8 [0, 0, 1, 1, 1, 0, 0, 1]
  , V 8 [1, 1, 0, 1, 1, 1, 0, 1]
  , V 8 [0, 0, 1, 1, 1, 0, 1, 0]
  , V 8 [0, 0, 1, 0, 0, 1, 1, 1]
  ]

scrambler = M 6 6 $ V 6
  [
    V 6 [1, 1, 0, 1, 1, 0]
  , V 6 [1, 0, 1, 1, 1, 1]
  , V 6 [0, 0, 1, 0, 1, 1]
  , V 6 [0, 0, 1, 1, 0, 0]
  , V 6 [1, 1, 0, 0, 0, 0]
  , V 6 [1, 1, 1, 0, 0, 0]
  ]

permutation = M 8 8 $ V 8
  [
    V 8 [0, 0, 0, 0, 0, 0, 0, 1]
  , V 8 [0, 0, 0, 0, 0, 0, 1, 0]
  , V 8 [0, 0, 0, 0, 0, 1, 0, 0]
  , V 8 [0, 1, 0, 0, 0, 0, 0, 0]
  , V 8 [0, 0, 0, 1, 0, 0, 0, 0]
  , V 8 [0, 0, 1, 0, 0, 0, 0, 0]
  , V 8 [0, 0, 0, 0, 1, 0, 0, 0]
  , V 8 [1, 0, 0, 0, 0, 0, 0, 0]
  ]

-- | 1 0 0 0 0 1 1 0 |
-- | 0 0 0 1 0 1 1 0 |
-- | 0 0 1 0 0 1 0 0 |
-- | 0 0 1 0 0 1 1 1 |
-- | 0 0 1 0 1 0 0 1 |
-- | 1 1 1 1 1 1 0 1 |
parity' = (scrambler `matProduct` parity) `matProduct` ↩
    permutation
```

Fig. 18. Niederreiter public and private keys.

```
-- | 0 1 0 0 0 1 |
ctext' = transpose $ scramblerInv `matProduct` (transpose↩
     ctext)

-- (z^2)*X + (z)
syndrome = goppaSyndromePoly (matToVec ctext') f g

-- (z^2)*X^2 + X + (z^2 + 1)
errorLocator = pattersonsAlgorithm x g syndrome

-- | 0 1 0 0 0 1 0 0 |
ctextError = V 8 $ map' (\c -> if 0 == evalPoly ↩
    errorLocator c then 1 else 0) codeLocators

-- | 0 0 1 0 0 0 1 0 |
ptext' = matToVec $ transpose $ permutationInv `↩
    matProduct` (transpose $ matFromVec ctextError)
```

Fig. 20. Niederreiter decryption.

V. Results

In this project we implemented the McEliece and Niederreiter cryptosystems in Haskell and verified them with refinement types using LiquidHaskell. This implementation included implementing the supporting matrix and Galois field math, as well as implementing variants of the Extended Euclid algorithm and Patterson's decoding algorithm. All of

the code written was verified with refinement types except for binary Galois field reduction and polynomial division which were not verified to terminate.

As part of this we implemented a size-aware API for vectors and matrices which provides bounds checking and prevents off-by-one errors.

During this project we found that refinement types can eliminate certain classes of errors when implementing cryptosystems with only a modest increase of in lines of code as shown by Figure 21.

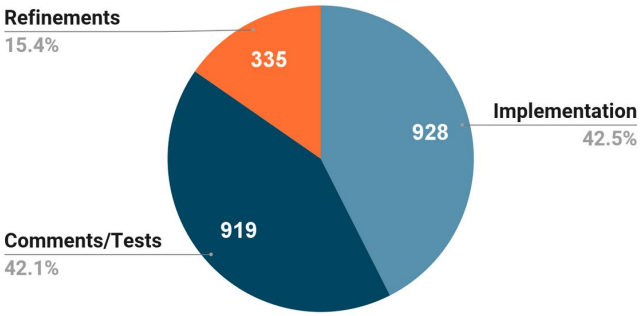

Fig. 21. Lines of code by component.

VI. Related Work & Acknowledgments

This project was inspired by MSR-INRIA's Project Everest which is implementing a formally verified TLS stack (https://project-everest.github.io/). The linear algebra package was based on the excellent LiquidHaskell tutorial at https://ucsd-progsys.github.io/liquidhaskell-tutorial/Tutorial_01_Introduction.html. Implementation of the McEliece and Niederreiter schemes was made possible through the use of the SAGE computer algebra system and by referencing papers by Risse, Roering, Minihold, and a SAGE implementation of McEliece by David Hu at https://github.com/davidhoo1988/Code_Based_Cryptography_Python.

VII. Conclusion

Refinement types have been a useful tool for implementing McEliece variants during this project. In many cases the refinements were simply assertions that would normally be checked at runtime, lifted into compile-time checks. Adding refinements was generally a one-time development cost that paid for itself when editing the code later. Combined with a good test suite (this project used doctests) this strategy could be used to build high-assurance systems.

The main challenge of working with refinement types was proving properties about custom data types, e.g. proving that dividing by a non-zero polynomial was safe, or proving that adding binary field elements of the same degree would always produce an element of a lower degree. Refinements also slightly increased the compile time for the project, although upgrading to the latest Z3 theorem prover noticeably improved this.

Future work could consist of verifying that polynomial division and binary Galois field reduction terminate. Additionally it would be useful to implement and verify key generation for the McEliece and Niederreiter schemes instead of relying upon SAGE code.

References

[1] Risse, T. (2011). How SAGE Helps To Implement Goppa Codes and The McEliece Public Key Crypto System. Hochschule Bremen, University of Applied Sciences.

[2] Roering, Christopher, "Coding Theory-Based Cryptopraphy: McEliece Cryptosystems in Sage" (2013). Honors Theses, 1963-2015. 17. https://digitalcommons.csbsju.edu/honors_theses/17

[3] NIST Computer Security Division. Post-quantum cryptography: CSRC. Retrieved February 24, 2022, from https://csrc.nist.gov/Projects/post-quantum-cryptography

[4] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17). Association for Computing Machinery, New York, NY, USA, 1789–1806. DOI:https://doi.org/10.1145/3133956.3134043

[5] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: experience with refinement types in the real world. SIGPLAN Not. 49, 12 (December 2014), 39–51. University of California San Diego.

[6] McEliece, R.J. (1978). A public key cryptosystem based on algebraic coding theory.

[7] Types and typeclasses. Types and Typeclasses - Learn You a Haskell for Great Good!. Retrieved April 22, 2022, from http://learnyouahaskell.com/types-and-typeclasses

[8] A Gentle Introduction to Haskell, Version 98. A gentle introduction to Haskell: Numbers. Retrieved April 22, 2022, from https://www.haskell.org/tutorial/numbers.html

[9] Pascal Véron. Code based cryptography and steganography. CAI 2013, 5th International Conference on Algebraic Informatics, Sep 2013, Porquerolles, France. pp.9-46.

[10] Minihold, Matthias. "Linear Codes and Applications in Cryptography." (2013). Master's Thesis, Vienna University of Technology.

[11] Stinson, D. R. (2019). Algorithm 6.2: Extended Euclid Algorithm. Cryptograph: Theory and Practice (4th ed., pp. 191). CRC Press.

[12] Classic mceliece: Intro. Classic McEliece: Intro. Retrieved April 23, 2022, from https://classic.mceliece.org/

[13] Jhala, R., Vazou, N.. (2020). Refinement Types: A Tutorial. https://arxiv.org/pdf/2010.07763v1.pdf

[14] William A. Stein et al. Sage Mathematics Software (Version 9.0), The Sage Development Team, 2020, http://www.sagemath.org.