# Verified Post-Quantum Cryptography

**Tom Arnold** <tca4384@rit.edu> | Advisor: **Dr. Stanisław Radziszowski** | **https://tom9729.bitbucket.io/csci788/**

*2022/04/14*

## Introduction

Post-quantum cryptography is being developed and standardized to resist attacks from quantum computers. Standardization focuses on correctness of the algorithm and performance of the implementation [2]. Formal verification techniques can be used to ensure correctness of the implementation to reduce the chance of implementation flaws breaking the security of the system.

McEliece is a post-quantum public-key cryptosystem based on binary Goppa codes and coding theory. Several variants of McEliece exist, one of which is a candidate in the NIST PQC competition (Classic McEliece). In this project we implemented McEliece and Niederreiter (a McEliece variant) in Haskell and verified the implementations using refinement types. The goal of the project was to show that refinement types are a useful tool for cryptography.

## McEliece / Niederreiter Cryptosystems

McEliece and Niederreiter are code-based public-key cryptosystems that are resistant to quantum attacks when implemented with a binary Goppa code and sufficiently large security parameters. The systems have equivalent security but Niederreiter can have 50% smaller keys and faster encryption (due to the smaller key) [4].

*Example: Niederreiter PKE [5]*

**Key Generation**

$n = 2^3 = 8$ $f = z^3 + z + 1$ $g = X^2 + X + 1$ codelocators = 0:1:[$(z^2)^i$ | i <- 1..6]

$$\left.\begin{array}{l} H = \begin{pmatrix} 1&0&0&1&0&0&0&1\\ 0&0&0&1&0&1&1&1\\ 0&0&1&1&1&0&0&1\\ 1&1&0&1&1&1&0&1\\ 0&0&1&1&1&0&1&0\\ 0&0&1&0&0&1&1&1 \end{pmatrix} \quad S = \begin{pmatrix} 1&1&0&1&1&0\\ 1&0&1&1&1&1\\ 0&0&1&0&1&1\\ 0&0&1&1&0&0\\ 1&1&0&0&0&0\\ 1&1&1&0&0&0 \end{pmatrix} \quad P = \begin{pmatrix} 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&1&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 1&0&0&0&0&0&0&0 \end{pmatrix} \end{array}\right\} \text{Private Key}$$

$H' = S \times H \times P$ } Public Key

**Encryption**

ptext = $\begin{pmatrix}0&0&1&0&0&0&1&0\end{pmatrix}$ ctext = H' x $\text{ptext}^T$ = $\begin{pmatrix}1&1&1&0&1&1\end{pmatrix}^T$

**Decryption**

syndrome = encodeAsPoly($S^{-1}$ x $\text{ctext}^T$) = $z^2X + z$

$(g_0, g_1)$ = split(g) | $(t_0, t_1)$ = split(t + X)
w = $g_0$ x inverse($g_1$, g) | r = $t_0$ + w x t1
t = polyInv(syndrome, g) | (_, u, v) = modifiedEEA(g, r)
} Patterson's algorithm [1]

errorlocator = $u^2 + X \times v^2 = z^2X^2 + X + z^2 + 1$

error = [errorlocator(c) | c <- codelocators] = $\begin{pmatrix}0&1&0&0&0&1&0&0\end{pmatrix}$

ptext' = $P^{-1}$ x $\text{error}^T$ = $\begin{pmatrix}0&0&1&0&0&0&1&0\end{pmatrix}^T$

## Refinement Types

Refinement types use logical predicates to make types more specific so that logic errors can be caught at compile time instead of runtime. Most programming languages have an integer type for example, but refinement types allow the developer to trivially annotate a function as accepting a subset of integers, e.g. only positive integers or only integers in a certain range. Refinement types allow reasoning about sizes and can be used to safely eliminate runtime bounds checking. Refinement types can even be used to verify that a function terminates.

For our verification work we used a Haskell implementation of refinement types called Liquid Haskell. This is a plugin for the GHC Haskell compiler which integrates refinement types via code annotations. The refinement types are checked at compile-time using the Z3 SMT solver which is installed separately [3].

*Example: Extended Euclid Algorithm*

```
{-@ 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'
```

Called with non-zero divisor so DBZ is unchecked.

1. **b$_0$** is greater than **r.**
2. Remainder is smaller than **r.**
3. **r** is decreasing.
4. ∴ function terminates. ∎

*Example: Appending Matrix Columns*

Inputs have same number of rows.

Output has same number of rows as input & the sum of the columns.

```
{-@ 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
```

*Example: Derived Constraints*

**d**-degree polynomial has **d + 1** coefficients.

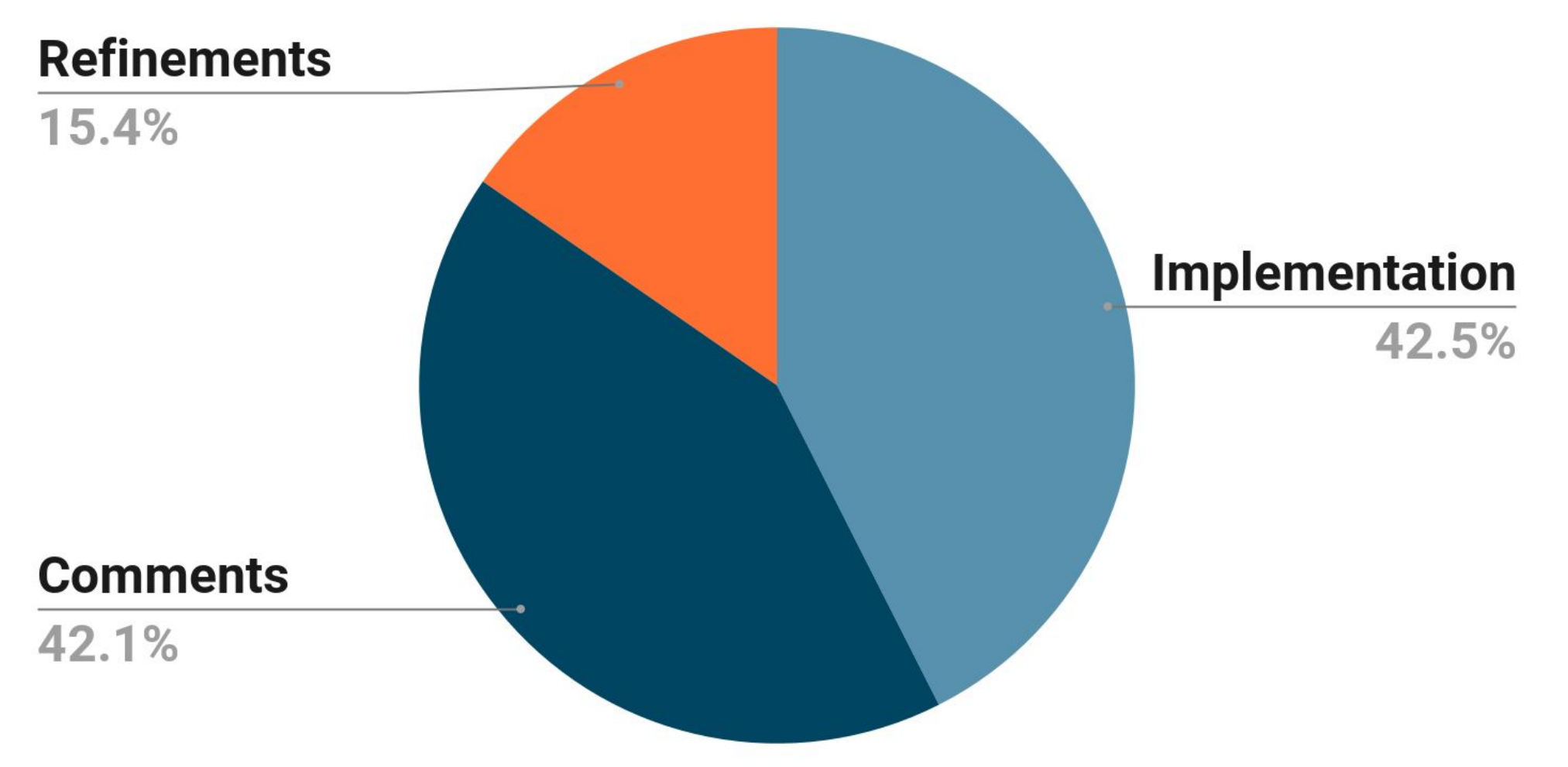
```
{-@ data Poly a = P { pDeg :: Nat
                    , pCoefs :: VectorN a {pDeg + 1} } @-}

{-@ parts :: ListN (List Bit) nParts @-}
parts = partitionList elts nParts
fElts = mkFieldElts parts

syndPoly = P (nParts - 1) $ V nParts fElts
```

Function returns list of size **nParts**.

Constructor refinement checks that degree & coefficients match up.

## Results

- Implemented McEliece and Niederreiter cryptosystems in Haskell and verified with refinement types.
  - Matrix and vector math including binary matrix inversion.
  - Univariate polynomial math and binary Galois fields.
  - Extended Euclid Algorithm.
  - Patterson's decoding algorithm for Goppa codes.
  - Toy examples of McEliece and Niederreiter encryption/decryption.
- **2,182** lines of code.
  - 928 lines of implementation.
  - 919 lines of comments & doctests.
  - 335 lines of refinement annotations.
- All code verified with refinement types *except* for GF2 reduction and polynomial division which were not verified to terminate.
  - Size-aware API for vectors and matrices provides bounds checking and prevents off-by-one errors.
  - Recursive functions guaranteed to terminate.
  - Avoiding divide by zero by requiring non-zero parameters.



## 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 working with refinement types was in 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 SMT solver has noticeably improved this.

## References

*This project was inspired by MSR-INRIA's Project Everest which is implementing a formally verified TLS stack (https://project-everest.github.io/).*

**[1]** Risse, T. (2011). How SAGE Helps To Implement Goppa Codes and The McEliece Public Key Crypto System.

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

**[3]** 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.

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

**[5]** Minihold, Matthias. "Linear Codes and Applications in Cryptography." (2013).