Verified Post-Quantum Cryptography

Milestone 3 - 2022/03/29

Tom Arnold <tca4384@rit.edu>

Problem Statement / Solution

Problem: Can we formally verify a post-quantum cryptosystem?

Solution:

- Implement variants of McEliece post-quantum cryptosystem.
- Formally verify implementation using refinement types (LiquidHaskell).
- Show benefits/tradeoffs of using refinement types in a small-scale project.

Milestone 3 Results

- ✅ Verified polynomial and Galois field implementations from milestone 2.
- ✅ Implemented Patterson's decoding algorithm.
- ✅ Implemented Niederreiter version of McEliece.

<u>Entire project is verified with refinement types.</u>

- 2,381 lines of code
- 8,882 constraints checked by LiquidHaskell (including inferred constraints)

Niederreiter Cryptosystem

- Proposed in the 80s (McEliece was 70s) [1].
- Same security as McEliece but more efficient (~50% key size, 10x fewer operations during encryption).
- Not probabilistic & can be used for digital signatures unlike McEliece.
- Similar implementation to McEliece except:
  - Parity matrix used instead of generator.
  - Plaintext is syndrome, i.e. message is encoded as errors in a message of zeroes.
- Like McEliece can be implemented with different codes but Goppa codes are the most secure.
- NIST PQC candidate "Classic McEliece" is based on Niederreiter [2].

Niederreiter Implementation

Bob (key generation)

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

t = 2

Private Key

```
parity = | 1 0 0 1 0 0 0 1 |   scrambler = | 1 1 0 1 1 0 |
         | 0 0 0 1 0 1 1 1 |               | 1 0 1 1 1 1 |
         | 0 0 1 1 1 0 0 1 |               | 0 0 1 0 1 1 |
         | 1 1 0 1 1 1 0 1 |               | 0 0 1 1 0 0 |
         | 0 0 1 1 1 0 1 0 |               | 1 1 0 0 0 0 |
         | 0 0 1 0 0 1 1 1 |               | 1 1 1 0 0 0 |

permutation = | 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 |
```

Parity derived from codelocators, scrambler/permutation random.

Public Key

```
parity' = scrambler x parity x permutation
```

Alice (encrypt)

Message encoded as weight t vector.

```
ptext = | 0 0 1 0 0 0 1 0 |
ctext = parity' x ptext^T = | 1 1 1 0 1 1 |^T
```

Bob (decrypt)

```
syndrome = encodeAsPoly(scrambler^-1 x ctext^T)
         = z^2X + z

(g0,g1)  = split(g)
w        = g0 x inverse(g1,g)
t        = polyInv(syndrome, g)
(t0,t1)  = split(t + X)
r        = t0 + w x t1
(_,u,v)  = modifiedEEA(g,r)

errorlocator = u^2 + X x v^2
             = z^2X^2 + X + z^2 + 1
```

Patterson's algorithm [3]

```
error = [errorlocator(c)| c <- codelocators]
      = | 0 1 0 0 0 1 0 0 |

ptext' = permutation^-1 x error^T
       = | 0 0 1 0 0 0 1 0 |^T
```

Refinement Examples

Function takes a list of values "xs" and a number of groups "n" and splits it into "n" many list of lists.

```
-- | Split a list into partitions of a given size.
--
-- Examples:
--
-- >>> partitionList [1,2,3,4,5] 2
-- [[1,2,3],[4,5]]
-- >>> partitionList [1,2,3,4,5] 3
-- [[1,2],[3,4],[5]]
{-@ partitionList :: xs:List a -> n:Pos -> ListN (List a) n @-}
```

Refinement Examples (2)

Refinement on data type checks that # of coefficients - 1 = polynomial degree.

```
-- Split the bits into several lists. Each sub-list will become
-- a field element coefficient of the syndrome polynomial.
{-@ parts :: ListN (List Bit) nElts @-}
parts = partitionList elts'' nElts

fElts = map' (\xs ->
                let xs' = dropWhile' (==0) xs
                    n = size xs'
                in E f $ if n == 0
                         then P 0 $ V 1 [0]
                         else P (n - 1) $ V n xs') parts

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

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

Roadmap For Project Completion

- Improve verification by checking additional properties.
  - Prove GF2 reduction terminates.
  - Prove EEA terminates.
- Work on poster and final report.

References

1. Cryptanalysis of the Original McEliece Cryptosystem
2. Classic McEliece
3. HOW SAGE HELPS TO IMPLEMENT GOPPA CODES AND THE McELIECE PUBLIC KEY CRYPTO SYSTEM