Verified Post-Quantum Cryptography

Verifying The McEliece Cryptosystem With Refinement Types

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

2022/04/28

Introduction

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

Classic McEliece is a post-quantum public-key cryptosystem being standardized by NIST which is based on binary Goppa codes and coding theory. Several variants of it exist; in this project we implemented two of them 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 = $0:1:[(z^2)^i \mid i \leftarrow 1..6]$

$H = \begin{pmatrix} 1&0&0&1&0&0&0&1 \\ 0&0&0&1&0&1&1&1 \\ 0&0&1&1&1&0&0&1 \\ 1&1&0&1&1&1&0&1 \\ 0&0&1&1&1&0&1&0 \\ 0&0&1&0&0&1&1&1 \end{pmatrix}$ $S = \begin{pmatrix} 1&1&0&1&1&1&1&1 \\ 1&0&1&1&1&1&1&0 \\ 0&0&1&0&1&1&0&0 \\ 0&0&1&1&0&0&0&0 \\ 1&1&0&0&0&0&1&0 \\ 1&1&1&0&0&0&0&0 \end{pmatrix}$ $P = \begin{pmatrix} 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \end{pmatrix}$ } Private Key

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

Encryption

ptext = $(0\ 0\ 1\ 0\ 0\ 0\ 1\ 0)$ ctext = $H' \times \text{ptext}^T = (1\ 1\ 1\ 0\ 1\ 1)^T$

Decryption

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

$(g_0, g_1) = \text{split}(g)$	$(t_0, t_1) = \text{split}(t + X)$
$w = g_0 \times \text{inverse}(g_1, g)$	$r = t_0 + w \times t1$
$t = \text{polyInv}(\text{syndrome}, g)$	$(\_, u, v) = \text{modifiedEEA}(g, r)$

} Patterson's algorithm [1]

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

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

ptext' = $P^{-1} \times \text{error}^T = (0\ 0\ 1\ 0\ 0\ 0\ 1\ 0)^T$

Note: This is a toy example for demonstration purposes. To provide comparable security to RSA-2048 this would require n=2048 and Goppa polynomial degree of 30 [4].

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. Refinement types allow the developer to annotate a function as accepting a subset of a type, 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 also be used to verify that a function terminates.

For our verification work we used Haskell (https://www.haskell.org/), a functional programming language with an advanced type system, and an extension called Liquid Haskell which adds support for refinement types via code annotations [3]. The refinement types are checked at compile-time using the Z3 theorem prover (https://github.com/Z3Prover/z3).

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. $\mathbf{b_0}$ 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.

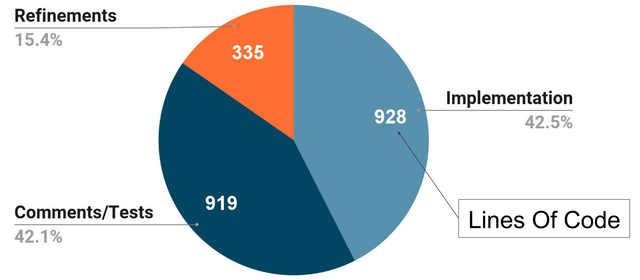
```
{-@ data Poly a = P { pDeg :: Nat
                    , pCoefs :: VectorN a {pDeg + 1} } @-}

{-@ 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.
- 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.
  - Avoid divide by zero by requiring non-zero parameters.
- Refinement types can eliminate certain classes of errors with only a modest increase of implementation complexity and are a useful tool for cryptography.



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.

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. Hochschule Bremen, University of Applied Sciences.

[2] NIST 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. University of California San Diego.

[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). Master's Thesis, Vienna University of Technology.