Verified Post-Quantum Cryptography

Milestone 2 (extended slides) - 2022/03/03

Tom Arnold <tca4384@rit.edu>

Problem Statement / Solution

Problem: PQC is important. 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.

Previous & Current Milestone Goals

- Milestone 1
  - ✅ Implement Hamming code version of McEliece.
  - ✅ Verified using refinement types.
- Milestone 2
  - ☐ Implement matrix inversion routine.
  - ☐ Research and implement Goppa code version of McEliece.

Milestone 2 Results

- ✅ Implemented and verified binary-matrix inversion.
  - Algorithm based on elementary-row operations.
  - Used during McEliece decryption, these were calculated by hand previously.
- ✅ Implemented Goppa code version of McEliece.
  - Implemented polynomial arithmetic and binary Galois field arithmetic.
  - **Operations**: addition, subtraction, multiplication, division, polynomial and field inversion, & polynomial evaluation.
  - **Operands**: polynomials of binary field elements.
  - **Not fully verified yet**, verification of polynomial arithmetic difficult.

~1000 lines of code written for this milestone.

Roadmap For Milestone 3

- Finish verification of polynomial and field arithmetic.
- Implement the Niederreiter variant of McEliece.

Polynomial Implementation

- Polynomial = vector of coefficients   [1 1 1] = [X^2 + X + 1]
- Degree of polynomial = size of vector
  - Tricky part: leading zero coefficients   [0 1 1 1] = [X^2 + X + 1], degree 2 not 3!
  - Doing this convenient for addition/subtraction, causes problems with everything else though
- Add/subtract: +/- coefficients
- Multiplication
  - Multiply pairs of terms, sum result
  - Degree of output = sum of degrees of input

```
 2 1 0     1 0      3      2      2     1     1     0
[A B C] x [D E] = [AD + AE + BD + BE + CD + CE]
```

[AX^2 + BX + C] x [DX + E] = [ADX^3 +AEX^2 + BDX^2 + BEX + CDX + CE]

```
                              3     2      1     0
                           = [AD  AEBD  BECD  CE]
```

Polynomial Implementation (2)

- Division
  - Quotient is 0, remainder is numerator
  - Divide leading terms -> Add to quotient, multiply by denominator and subtract from remainder
  - Stop when remainder is zero or degree of remainder is less than denominator
  - Degree of outputs
    - Quotient: difference of leading terms if numerator has larger degree, otherwise 0
    - Remainder: degree of second term of denominator if smaller, otherwise degree of denominator

$n = X^5 + 1$
$d = X^3$

$q = 0$
$r = X^5 + 1$
$t = X^5 / X^3 = X^2$
$q' = X^2$
$r' = X^5 + 1 - (X^2 \times X^3) = 1$

$deg(r') < deg(d)$ so return $(q, r) \to (X^2, 1)$

https://en.wikipedia.org/wiki/Polynomial_long_division#Pseudocode

Polynomial Implementation (3)

- Modular inverse (p mod g)
  - Run EEA on p and g: (b, s, t) = eea(p, g)
  - Divide S by leading term of B: s / lead(b)

http://juaninf.blogspot.com/2013/04/function-make-div-with-id-mycell-sage.html

Galois Field Implementation

- Field = polynomial of bits (splitting/irreducible polynomial)
- Element = field and a polynomial of bits
- Add/subtract: same as polynomial (bit type is mod 2)
- Multiply: same as polynomial except reduce if result degree >= field polynomial
- Reduce
  - Add 0's to field polynomial on the right so the degrees of both line up, then add them
  - Repeat while degree >= field polynomial

Field = $X^3 + X^2 + X + 1$ = [1 1 1 1]

$[X^2 + X + 1]$ x $[X^2 + 1] = [X^4 + X^3 + X + 1]$ = [1 1 0 1 1]	deg >= 3
1 1 1 1 0	add field polynomial shifted left 1
1 0 1	result has degree 2 < 3 so done
= $[X^2 + 1]$	

Galois Field Implementation (2)

- Division of polynomials of bits: same as polynomial except AND coefficients instead of dividing
- Division of field elements: find inverse via EEA and multiply

McEliece w/ Goppa Code Implementation

Bob (setup)

```
n = 2^3 = 8
f = z^3 + z + 1
g = X^2 + X + 1

codelocators = 0:1:[z^i | i <- 1..6]
syndlocators = [(X - c)^-1 % g | c <- codelocators]
```

Generator derived from codelocators, scrambler/permutation random.

```
generator = |1 1 0 0 1 0 1 1|   scrambler = |0 1|
            |0 0 1 1 1 1 1 1|               |1 0|

permutation = |0 1 0 0 0 0 0 0|
              |0 0 0 1 0 0 0 0|
              |0 0 0 0 0 0 1 0|
              |1 0 0 0 0 0 0 0|
              |0 0 1 0 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 0 1|

generator' = scrambler x generator x permutation
```

Alice (encrypt)

```
ptext = |1 0|
err   = |0 0 0 0 1 0 0 0|
ctext = ptext x generator' + err
      = |1 0 1 0 0 1 1 1|
```

Bob (decrypt)

```
ctext'  = ctext x permutation^-1
ctext'' = encodeAsVecPoly(ctext')
        = |0 0 1 1 1 1 0 1|

syndrome = dotProd(ctext'', syndlocators)
         = (z)X + (z^2 + z + 1)

elp = extEuclidAlg(g, syndrome)
    = (z^2 + 1)X + (z^2 + z)

err = [if elp(c) == 0 then 1 else 0 | c <- codelocators]
    = |0 0 0 0 0 0 1 0|

ctext''' = ctext'' + err
         = |0 0 1 1 1 1 1 1|

ctext'''' = ctext''' x generator^-1
          = |0 1|

ptext = ctext'''' x scrambler^-1
      = |1 0|
```

References

- The Theory Of Error Correcting Codes chapter 12, Macwilliams and Sloane
- How Sage Helps To Implement Goppa Codes And The McEliece Public Key Crypto System, Risse
- Coding Theory-Based Cryptography: McEliece Cryptosystems In Sage, Roerin
- Code Based Cryptography in Python, David J.W. Hu
- Ejemplo Criptosistema McEliece en SAGE, Juan Grados
- A Course in Computational Algebraic Number Theory chapter 3, Henri Cohen
- Polynomial long division - Wikipedia