A performance analysis of NTRUencrypt and Shamir's Secret Sharing

Michael Chen, Kyle Ng, Victor Zheng

NTRUEncrypt

- Public Key Cryptosystem
- Lattice-based alternative to RSA
  - Resistant to quantum computers
- Built around difficulty of factoring polynomials in a ring into a quotient of two small polynomials

Public Parameters & Other Notes

- *N* - Degree of polynomial
  - Used to form polynomial ring - (X^*N* - 1)
- *p* - Modulus of “small” coefficients
- *q* - Modulus of “large” coefficients
- Assume *N* is prime, *q* > *p*, *q* and *p* are coprime
- All polynomial calculations done mod (X^*N* - 1)
- Modulus may be “centered” on zero
  - 2 (mod 3) = -1 (mod 3)
- Integer modulus on polynomials means apply modulus to the coefficients

Key Generation

1. Generate two polynomials *f, g* with coefficients (mod *p*) centered
2. Let *fp* and *fq* be the inverse of *f* (mod *p*) and (mod *q*)
3. Let $h = pf_q \cdot g (\mathrm{mod}\ q)$

Then *h* is the public key and *f,fp* are the private keys

Key Generation - Example

- Assume parameters are *N*=7, *p*=3, *q*=16
- Now we generate two polynomials

$$f = 1 + X^3 - X^6$$

$$g = -X^3 + X^6$$

Then we can compute *fp* and *fq*

$$f_p = X + X^2 + 2X^3 + 2X^4 + X^6 (\text{mod} 3)$$

$$f_q = 13 + 13X + 11X^2 + 4X^3 + X^4 + 14X^5 + 9X^6 (\text{mod } 16)$$

- Finally we can compute *h*

$$h = pf_q \cdot g (\text{mod } q) = 4 + 7X + X^2 - 4X^3 + 3X^4 - 6X^5 - 5X^6 (\text{mod } 16)$$

Encryption

- Encode the message *m* to send as a polynomial with coefficients (mod *p*) centered
- Randomly generate a polynomial *r* with small coefficients
- Let $e = r \cdot h + m (\mathrm{mod}\ q)$

Encryption Example

- Assume our message polynomial is:

$$m = -1 + X^3 - X^4$$

- Assume our random polynomial is:

$$r = X - X^5$$

- Then our encrypted polynomial is:

$$e = r \cdot h + m (\text{mod } q) = 9 + 8X + 4X^2 + 8X^3 + 15X^5 + 3X^6 (\text{mod } 16)$$

Decryption

- Compute a = f * e (mod q) centered
  - Must compute centered mod since original message was centered
- Compute b = a (mod p) centered
- Finally, compute m = fp * b (mod p) centered

Decryption Example

- First we compute *a*

$$a = f \cdot e (\mathrm{mod}\ q) = 1 + 3X - X^2 + X^3 - 7X^4 + 2X^6 (\mathrm{mod}\ 16)$$

- Next we compute *b*

$$b = a (\mathrm{mod}\ 3) = 1 - X^2 + X^3 - X^4 - X^6 (\mathrm{mod}\ 3)$$

- Finally we can compute *m*

$$m = f_p \cdot b (\mathrm{mod}\ p) = -1 + X^3 - X^4 (\mathrm{mod}\ 3)$$

- Which is our original message

Performance Results

keygen	encrypt	decrypt	N	q	p
35.53 ms	0.51 ms	1.01 ms	167	128	3
82.4 ms	0.94 ms	1.88 ms	251	128	
156.55 ms	1.62 ms	3.44 ms	347	128	
320.82 ms	3.06 ms	6.18 ms	503	256	

Performance Results (continued)

keygen	encrypt	decrypt	N	q	p
0.94 ms	0.03 ms	0.027 ms	11	32	3
109.48 ms	0.92 ms	1.88 ms	250		
475.54 ms	2.97 ms	6.07 ms	500		
1.51 s	6.25 ms	12.67 ms	750		
2.28 s	10.1 ms	20.91 ms	1000		

Performance Results (continued)

keygen	encrypt	decrypt	N	q	p
20.43 ms	0.42 ms	0.82 ms	100	32	3
21.62 ms	0.42 ms	0.74 ms		64	
21.51 ms	0.41 ms	0.74 ms		128	
21.07 ms	0.41 ms	0.76 ms		256	
21.45 ms	0.43 ms	0.75 ms		512	

Conclusion

- High performance public-key cryptosystem resistant to quantum attacks
- Care needs to be taken with generating f and g
- Care needs to be taken with encoding messages

Secret Sharing Made Short (SSMS)

- Method of breaking up secret to participants
- Implemented based on Shamir's Secret Sharing Scheme, Rabin's Information Dispersal Algorithm, and AES

Shamir’s Secret Sharing Scheme (SSSS)

- Obtains secret and produces *n* shares, m of which are required to recover secret
- For SSMS, we divided the private key into n keys
- Shares of key are distributed to participants
- Shares are the same size as secret
- Builds polynomial; m points are required to determine graph

Example of SSSS

```
Test Shamir's
Secret: 872600351619698574554382914956576642l4
Shares:
1, 126679520209216791504301063790908566114
2, 453850173342512769868562619210127330O5
3, 135177099975425456347911896018542706l4
4, 310775981990905974481058468334331789l1
5, 980646819388954324268002336l5749457986
6, 443377777564878188391870462329190020l2

Recovered: 872600351619698574554382914956576642l4
```

Advanced Encryption Standard (AES)

- Using Cipher Block Chaining (CBC)
- IV and Key are generated
- Enc(plaintext, IV, key) = ciphertext
- Dec(ciphertext, key) = IV, plaintext

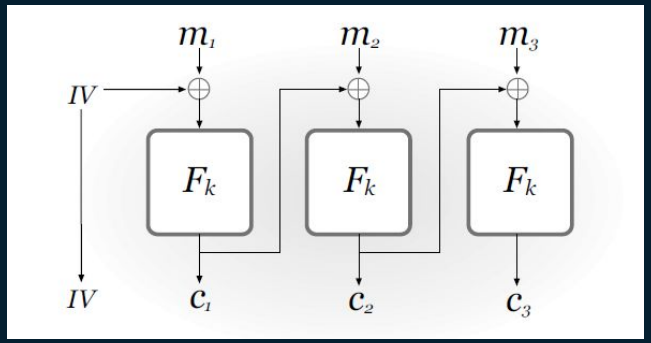


Figure 1: CBC diagram from KL

Rabin's Information Dispersal Algorithm (IDA)

- Splits information into *n* fragments, *m* of which is required to recover
- Fragments are smaller than information size
- For SMSS, we used IDA to distribute the ciphertext

Shamir's

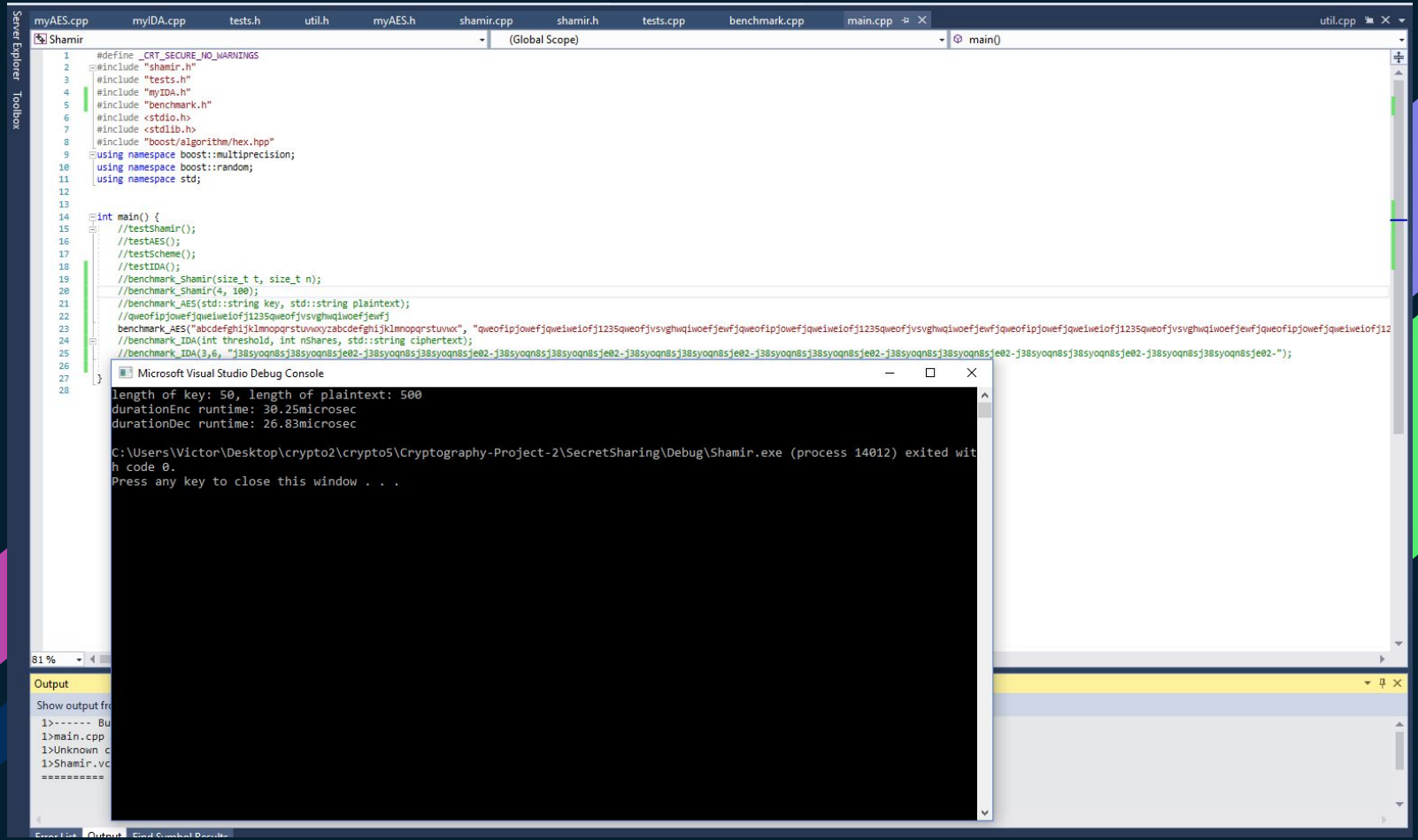
keygen&split	recover	#Digits	t	n
0.27584 ms	0.42278 ms	38	4	6
0.37871 ms	0.41122 ms		4	10
0.7983 ms	0.39483 ms		4	25
1.5079 ms	0.39757 ms		4	50
2.90139 ms	0.39811 ms		4	100

AES

encrypt	decrypt	keyLen	plaintextLen
0.02832 ms	0.02469 ms	50	50
0.02961 ms	0.02613 ms		100
0.03039 ms	0.02607 ms		200
0.03025 ms	0.02683 ms		500
0.03323 ms	0.02861 ms		1000

IDA

cipherSplit	ciphRecov	cipherLen	t	frag
7.197 ms	1.568 ms	10	3	6
7.517 ms	1.946 ms	25	3	6
8.635 ms	1.645 ms	50	3	6
8.222 ms	1.710 ms	100	3	6
9.129 ms	1.946 ms	200	3	6


myAES.cpp
myIDA.cpp
tests.h
util.h
myAES.h
shamir.cpp
shamir.h
tests.cpp
benchmark.cpp
main.cpp
util.cpp
Shamir
(Global Scope)
main()
#define _CRT_SECURE_NO_WARNINGS
#include "shamir.h"
#include "tests.h"
#include "myIDA.h"
#include "benchmark.h"
#include <stdio.h>
#include <stdlib.h>
#include "boost/algorithm/hex.hpp"
using namespace boost::multiprecision;
using namespace boost::random;
using namespace std;
int main() {
//testShamir();
//testAES();
//testScheme();
//testIDA();
//benchmark_Shamir(size_t t, size_t n);
//benchmark_Shamir(4, 100);
//benchmark_AES(std::string key, std::string plaintext);
//qweofipjowefjqweiweiofj1235qweofjvsvghwqiwoefjewfj
benchmark_AES("abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwx", "qweofipjowefjqweiweiofj1235qweofjvsvghwqiwoefjewfjqweofipjowefjqweiweiofj1235qweofjvsvghwqiwoefjewfjqweofipjowefjqweiweiofj1235qweofjvsvghwqiwoefjewfjqweofipjowefjqweiweiofj12
//benchmark_IDA(int threshold, int nShares, std::string ciphertext);
//benchmark_IDA(3,6, "j38syoqn8sj38syoqn8sje02-j38syoqn8sj38syoqn8sje02-j38syoqn8sj38syoqn8sje02-j38syoqn8sj38syoqn8sje02-j38syoqn8sj38syoqn8sje02-j38syoqn8sj38syoqn8sje02-j38syoqn8sj38syoqn8sje02-");
}
Microsoft Visual Studio Debug Console
length of key: 50, length of plaintext: 500
durationEnc runtime: 30.25microsec
durationDec runtime: 26.83microsec
C:\Users\Victor\Desktop\crypto2\crypto5\Cryptography-Project-2\SecretSharing\Debug\Shamir.exe (process 14012) exited with code 0.
Press any key to close this window . . .
81 %
Output
Show output from:
1>------ Build started: Project: Shamir, Configuration: Debug Win32 ------
1>main.cpp
1>Unknown compiler version - please run the configure tests and report the results
1>Shamir.vcxproj -> C:\Users\Victor\Desktop\crypto2\crypto5\Cryptography-Project-2\SecretSharing\Debug\Shamir.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
Error List
Output
Find Symbol Results
Server Explorer
Toolbox

Conclusion

- The modified Secret Sharing with Shamir's Secret Sharing + AES + Rabin's IDA is more secure and efficient