Optimizing Post-Quantum Cryptographic Algorithms for Modern and Future Processor Architectures

Shay Gueron
University of Haifa and Intel Corporation

Fabian Schlieker
Ruhr University Bochum
Horst Görtz Institute for IT-Security,

June 28, 2016
Outline

• Introduction
  • Post-Quantum Cryptography
  • Modern Processor Architecture Features
• Implementation
  • NewHope
  • NTRU
  • ring-TESLA
• Summary
Threat of Quantum Computers

• Shor’s Algorithm (1994) allows factorization, on a quantum computer, in polynomial time, instead of exponential time with best classical algorithms
• Other usages as well
• Effectively breaks all of today’s widely deployed public-key cryptography
  • RSA, DH, ECC
  • And symmetric crypto with a 128-bit secret
• However: no efficient quantum computer has been built yet
  • Heavily researched in academia, industry as well as by government agencies
  • Estimation by IBM (2012): 10—15 years
  • If and when: ??
Post-Quantum Cryptography

• Encrypted traffic today can be *stored* and then decrypted as soon as an efficient quantum computer exists

• A cryptosystem needs time to gain confidence
  • “ECC was introduced in 1985 but is only being widely deployed on the Internet since 2015”
    • (DJB & Tanja) [https://events.ccc.de/congress/2015/Fahrplan/events/7210.html](https://events.ccc.de/congress/2015/Fahrplan/events/7210.html)

→ We need to come up with a solution (alternatives) as soon as possible
  → *And we want such solutions to be efficient*

• There are promising directions
  • Cryptosystems can be built on different mathematical problems that are not susceptible to Shor’s algorithm (and other), to survive after PQ computer exists
    • “Post-Quantum Cryptography”
  • *Lattice-based*, Code-based, Hash-based, Multivariate Quadratics based
Our study

- Performance optimization of some Lattice-based schemes
- We addressed the basic primitives:
  - Encryption, signature, key exchange
- We chose:
  - NewHope (key exchange)
  - NTRU (encryption)
  - ring-TESLA (signatures)
- So, how faster can they go?
Outline

• Introduction
  • Post-Quantum Cryptography
  • Modern Processor Architecture Features
• Implementation
  • NewHope
  • NTRU
  • ring-TESLA
• Summary
**Single Instruction Multiple Data (SIMD)**

- “Vectorization”; do 4/8/16/32 times the work in only one instruction
- SSE (1999): 128-bit registers; e.g., compute 4 x 32 bit operations
- AVX (2011, Sandy Bridge): 256-bit registers, floating-point instructions, non-destructive destination
- AVX2 (2013, Haswell): integer instructions
- AVX512 (announced):
  - 512-bit registers and 32 of them
  - Mask operands for more powerful data-control
  - Many more new instructions

```c
char a[N], b[N], c[N];
__m128i *va = (__m128i*)a;
__m128i *vb = (__m128i*)b;
__m128i *vc = (__m128i*)c;

for (i = 0; i < N; i += 16) {
    __m128i rb = _mm_loadu_si128(&vb[i]);
    __m128i rc = _mm_loadu_si128(&vc[i]);
    __m128i ra = _mm_add_epi8(rb, rc);
    _mm_storeu_si128(&va[i], ra);
}
```
AES Native Instructions (AES-NI)

- Hardware instructions for extremely fast AES computation

```c
__m128i m = _mm_load_si128((const __m128i *) m);

m = _mm_xor_si128(m, K0);

m = _mm_aesenc_si128(m, K1);
...

m = _mm_aesenc_si128(m, K9);

m = _mm_aesenclast_si128(m, K10);

_mm_store_si128((__m128i *) _out, m);
```
Outline

• Introduction
  • Post-Quantum Cryptography
  • Modern Processor Architecture Features
• Implementation
  • NewHope
  • NTRU
  • ring-TESLA
• Summary
Post-Quantum Key Exchange

• “NewHope” (2015)
  • Highly optimized implementation by Alkim, Ducas, Pöppelmann and Schwabe
  • Smaller parameter choice; faster sampling; hand-crafted SIMD code
  • This is our baseline. We further improve parts of it
• Attracted increased attention recently; already implemented in
  • TOR handshake protocol
  • BoringSSL by Google
  • LatticeCrypto library by Microsoft
NewHope Protocol

Parameters: $q = 12289 < 2^{14}$, $n = 1024$

Error distribution: $\psi_{12}$

<table>
<thead>
<tr>
<th>Alice (server)</th>
<th>Bob (client)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{seed} \leftarrow {0, 1}^{256}$</td>
<td>$s', e', e'' \leftarrow \psi_{12}^n$</td>
</tr>
<tr>
<td>$a \leftarrow \text{Parse}(\text{SHAKE-128}(\text{seed}))$</td>
<td>$a \leftarrow \text{Parse}(\text{SHAKE-128}(\text{seed}))$</td>
</tr>
<tr>
<td>$s, e \leftarrow \psi_{12}^n$</td>
<td>$u \leftarrow as' + e'$</td>
</tr>
<tr>
<td>$b \leftarrow as + e \xrightarrow{(b, \text{seed})} (b, \text{seed})$</td>
<td>$v \leftarrow bs' + e''$</td>
</tr>
<tr>
<td>$v' \leftarrow us \xleftarrow{(u, r)} (u, r)$</td>
<td>$r \leftarrow \text{HelpRec}(v)$</td>
</tr>
<tr>
<td>$\nu \leftarrow \text{Rec}(v', r)$</td>
<td>$\nu \leftarrow \text{Rec}(v, r)$</td>
</tr>
<tr>
<td>$\mu \leftarrow \text{SHA3-256}(\nu)$</td>
<td>$\mu \leftarrow \text{SHA3-256}(\nu)$</td>
</tr>
</tbody>
</table>

- Polynomial multiplication $as, as', bs', us$ and add. of error vectors $e, e', e''$
  - Most computationally intensive part, but heavily optimized
- Function $\text{Parse()}$ to generate polynomial $a$ became new bottleneck and therefore is our optimization target
Optimization 1: Decreased rejection rate

- Original rejection sampling:
  - Two bytes are successively taken from the pseudorandom stream to form a candidate for each of the $n = 1024$ coefficients
  - Upper two bits are discarded and 14-bit value accepted when $< q$
  - Acceptance probability: $q / 2^{14} = 12289 / 16384 = 75\%$
Optimization 1: Decreased rejection rate

- Original rejection sampling:
  - Two bytes are successively taken from the pseudorandom stream to form a candidate for each of the \( n = 1024 \) coefficients
  - Upper two bits are discarded and 14-bit value accepted when < \( q \)
  - Acceptance probability: \( \frac{q}{2^{14}} = \frac{12289}{16384} = 75\% \)

- Our proposal: don’t discard bits!
  - sample from 16-bit range; \( \lfloor 2^{16}/q \rfloor = 5 \) thus accept when < 5\( q \)
  - Acceptance probability: \( \frac{5q}{2^{16}} = \frac{61445}{65536} = 94\% \)
  - Subsequently subtract \( q \) until in range \([0, q)\)
  - Result is still uniformly random in that range
Optimization 2: Vectorized sampling

- Vectorized rejection sampling:
  - Do not check each candidate one by one
  - Use AVX2 / AVX512 to check 16 / 32 candidates in parallel
  - Not exactly straightforward (to do efficiently) with AVX2
  - Much easier with AVX512 thanks to a handy new instruction
Optimization 3: Faster random generation

- Faster generation of pseudorandom bytes:
  - Pseudorandom stream is generated using the SHAKE-128 extendable output function (part of SHA-3)
  - We do not need preimage-/collision resistance (the seed is public anyway)
  - The desired property is just indistinguishability from random bytes
- There are faster alternatives to this function
  - AES-256 with a pipelined implementation in counter mode leveraging AES-NI
    - Counter value as plaintext, seed as key
    - Good approximation for a PRP, so ciphertext is also indistinguishable from random
Results (relative speedup)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Server</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>All 3 optimizations</td>
<td>1.59x</td>
<td>1.54x</td>
</tr>
</tbody>
</table>
Results (absolute numbers)

Table 1. The performance of the different optimizations, compared to ADPS [2] as the baseline. The numbers represent the cycles counts, measured using the test bench (lower is better) and the speedup factor compared to the baseline that is set to 1 (i.e., higher is better).

<table>
<thead>
<tr>
<th>Method</th>
<th>parse cycles</th>
<th>Server cycles</th>
<th>Server speedup</th>
<th>Client cycles</th>
<th>Client speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>47,044</td>
<td>113,361</td>
<td>1.13x</td>
<td>115,909</td>
<td>1.12x</td>
</tr>
<tr>
<td>I, II</td>
<td>38,466</td>
<td>100,343</td>
<td>1.27x</td>
<td>104,120</td>
<td>1.24x</td>
</tr>
<tr>
<td>I, II, III</td>
<td>32,080</td>
<td>94,183</td>
<td>1.36x</td>
<td>97,688</td>
<td>1.32x</td>
</tr>
<tr>
<td>I, II, IV</td>
<td>17,053</td>
<td>80,087</td>
<td>1.59x</td>
<td>84,119</td>
<td>1.54x</td>
</tr>
</tbody>
</table>
Outline

• Introduction
  • Post-Quantum Cryptography
  • Modern Processor Architecture Features

• Implementation
  • NewHope
  • NTRU
  • ring-TESLA

• Summary
NTRUEncrypt

- Proposed in 1998 by Hoffstein, Pipher and Silverman
- Based on hard problems on lattices
- Over time, attacks were found and parameters adjusted
- Standardized in 2008 (IEEE Std. 1363.1-2008)
- Open-source C implementation available since 2011
  - On GitHub by third party author (“Tim Buktu”)
  - Faster than reference implementation by the original authors
  - Has SSE vectorization
- Our work:
  - Extended vectorization to AVX2 and AVX512
  - Improved pseudorandom number generation using AES-NI
Porting SSE to AVX2 / AVX512

- Mostly straightforward
  - “128” → “256” / “512”
  - Double the number of elements processed in parallel
  - Halve the number of loop iterations
- Still, some pitfalls to handle
Using AES-NI to generate pseudorandomness

- Pseudo-random number generation done using SHA-1 and SHA-256
- At the time NTRUEncrypt was standardized, hash functions were a good way to produce uniformly/randomly distributed bytes
  - But nowadays very slow compared to AES block cipher, for which hardware instructions are available and whose output looks also randomly distributed
- We exchanged the SHA hash functions with a highly optimized and pipelined AES implementation based on AES-NI
## Benchmark results

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (SSE)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AVX2</td>
<td>1.23x</td>
<td>1.37x</td>
</tr>
<tr>
<td>AES-NI</td>
<td>1.36x</td>
<td>1.18x</td>
</tr>
<tr>
<td>Both</td>
<td>1.84x</td>
<td>1.76x</td>
</tr>
</tbody>
</table>
Predicting performance on future platforms

- Problem: no processors with AVX512 are available yet
- We used the Intel Software Development Emulator (SDE) to predict the instructions count on platforms that are not available in silicon
- The required # of instructions for decryption is ~50% using AVX512

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (SSE)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AVX2</td>
<td>0.79x</td>
<td>0.69x</td>
</tr>
<tr>
<td>AVX512</td>
<td>0.68x</td>
<td>0.53x</td>
</tr>
</tbody>
</table>
Outline

- Introduction
  - Post-Quantum Cryptography
  - Modern Processor Architecture Features
- Implementation
  - NewHope
  - NTRU
  - ring-TESLA
- Summary
Post-Quantum Digital Signatures

- “ring-TESLA” (2016) by Akleylek, Bindel, Buchmann, Krämer and Marson
- Also based on R-LWE; provably secure instantiation
- Built upon older code; potential for improvements
Optimization 1: FMA

- Fused-Multiply-Add
- Example:
  - \( a = b \times c + d \)
  - \( e = f \times g + h \)
  - \( i = j \times k + l \)
  - \( m = n \times o + p \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub</td>
<td>3</td>
</tr>
<tr>
<td>Mul</td>
<td>5</td>
</tr>
<tr>
<td>FMA</td>
<td>5</td>
</tr>
</tbody>
</table>

```
v = _mm256_mul_pd(vc, vq);
vx = _mm256_add_pd(vx, vc);
v = _mm256_fnmadd_pd(vc, vq, vx);
```
Optimization 2: Explicit Vectorization

- Don’t rely on auto-vectorization by the compiler
  - Because it doesn’t do it in this case 😞
- Explicitly vectorize coefficient-wise operations (mul, add, sub)

```c
for(i = 0; i < PARAM_N; i++)
{
    result[i] = (x[i] + y[i]) % PARAM_Q;
}
```

```c
for(i = 0; i < PARAM_N; i+=4)
{
    vx = _mm256_load_pd(x+i);
    vy = _mm256_load_pd(y+i);
    vt = _mm256_add_pd(vx, vy);

    vc = _mm256_mul_pd(vt, vqinv);
    vc = _mm256_round_pd(vc,0x08);
    vt = _mm256_fnmadd_pd(vc,vq,vt);
    _mm256_store_pd(result+i, vt);
}
```
Optimization 3: Code Rearrangement

- Avoid if-conditions in inner loops
- Arrange memory accesses sequentially

```c
for(i=0;i<PARAM_N;i++)
{
    for(j=0;j<PARAM_W;j++)
    {
        pos=pos_list[j]+i;
        if(pos>=PARAM_N){
            Ec[pos-PARAM_N] += e[i];
        }
        else{
            Ec[pos] -= e[i];
        }
    }
}
```

```c
for(j=0;j<PARAM_W;j++)
{
    pos=pos_list[j];
    for(i=0;i<pos;i++)
    {
        Ec[i] += e[i-pos+PARAM_N];
    }
    for(i=pos;i<PARAM_N;i++)
    {
        Ec[i] -= e[i-pos];
    }
}
```
## Results

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Sign</th>
<th>Verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Optimized</td>
<td>1.89x</td>
<td>1.78x</td>
</tr>
</tbody>
</table>
Outline

• Introduction
  • Post-Quantum Cryptography
  • Modern Processor Architecture Features
• Implementation
  • NewHope
  • NTRU
  • ring-TESLA
• Summary
Summary

- Optimized implementations of three important cryptographic primitives
  - Encryption, Key-Exchange, Signatures
  - All lattice-based schemes
- Optimized at three different layers:
  - Code level
    - SSE $\rightarrow$ AVX*; MUL+ADD $\rightarrow$ FMA
  - Architectural level
    - Rearrange loops while maintaining correct semantics
  - Algorithmic level
    - Discard less candidates in coefficient sampling, use fast crypto (AES-NI)
- Achieved significant speedups, even for the already very efficient code
- (Ideal-) Lattice-crypto can be very fast 😊
  - Promising efficient candidate for standardized post-quantum schemes… ?!
**Code availability**

- Code available at
  - [https://github.com/fschlieker/newhope](https://github.com/fschlieker/newhope)
  - [https://github.com/fschlieker/libntru](https://github.com/fschlieker/libntru)
  - [https://github.com/fschlieker/ring-TESLA](https://github.com/fschlieker/ring-TESLA)

- Publications:


Acknowledgements

- This research was supported by the PQCRYPTO project, which was partially funded by the European Commission Horizon 2020 research Programme, grant #645622.