Cryptographic Engineering

Cryptography in software – the basics (ctd.)

Radboud University, Nijmegen, The Netherlands

Spring 2019

“The multicore revolution”

- Until early years 2000 each new processor generation had higher clock speeds
- Nowadays: increase performance by number of cores:
  - My laptop has 2 physical (and 4 virtual) cores
  - Smartphones typically have 2 or 4 cores
  - Servers have 4, 8, 16,. . . cores
  - Special-purpose hardware (e.g., GPUs) often comes with many more cores
- Consequence: “The free lunch is over” (Herb Sutter, 2005)

> *“As a result, system designers and software engineers can no longer rely on increasing clock speed to hide software bloat. Instead, they must somehow learn to make effective use of increasing parallelism.”*
>
> —Maurice Herlihy: The Multicore Revolution, 2007

Why multicore doesn't matter…
… for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

- **If you perform only one crypto operation, you don't care**
- **Many crypto operations are trivially parallel on multiple cores**

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 4 consecutive 32-bit integers (a_0, a_1, a_2, a_3)
- Load 4 consecutive 32-bit integers (b_0, b_1, b_2, b_3)
- Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- Store 128-bit vector (c_0, c_1, c_2, c_3)

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most "large" processors
- Instructions for vectors of bytes, integers, floats...
- Need to interleave data items (e.g., 32-bit integers) in memory
- Compilers will not really help with vectorization

Back to adding up 1000 integers

- Imagine that
  - vector addition is as fast as scalar addition
  - vector loads are as fast as scalar loads
- Need only 250 vector additions, 250 vector loads ($+$ adding up 4 partial sums)
- Lower bound of 250 cycles
- Very straight-forward modification of the program
- Fully unrolled loop needs only $1/4$ of the space

Is it really that efficient?

- Consider the Intel Skylake processor with AVX2
    - 32-bit load throughput: 2 per cycle
    - 32-bit add throughput: 4 per cycle
    - 32-bit store throughput: 1 per cycle
    - 256-bit load throughput: 2 per cycle
    - 8× 32-bit add throughput: 3 per cycle
    - 256-bit store throughput: 1 per cycle
- **AVX2 vector instructions are almost as fast as scalar instructions but do 8× the work**
- Situation on other architectures/microarchitectures is similar
- Reason: cheap way to increase arithmetic throughput (less decoding, address computation, etc.)

More reasons for using vector arithmetic

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups
- Secret-data-dependent branches and secret branch conditions are the major sources of timing-attack vulnerabilities
- Strong synergies between speeding up code with vector instructions and protecting code!

Vectorization problems I

Carry handling

- When adding two 32-bit integers, the result may have 33 bits (32-bit result + carry)
- Scalar additions keep the carry in a special *flag register*
- Subsequent instructions can use this flag, e.g., "add with carry"
- How about carries of vector additions?
    - Answer 1: Special "carry generate" instruction (e.g., CBE-SPU)
    - Answer 2: They're lost, recomputation is expensive
- Need to *avoid carries* instead of handling them
- No problem for today's lecture, but requires care for big-integer arithmetic

Vectorization problems II

Removing instruction-level parallelism

- If we don't vectorize we perform multiple independent instructions
- We turn *data-level parallelism (DLP)* into *instruction-level parallelism (ILP)*
- Pipelined and multiscalar execution need ILP
- Vectorization removes ILP
- Problematic for algorithms with, e.g., 4-way DLP
- Good example to see this: ChaCha vs. Blake
- Vectorization of ChaCha can resort to higher-level parallelism (multiple blocks)
- Harder for Blake: each block depends on the previous one

Vectorization problems III

Data shuffeling

- Consider multiplication of 4-coefficient polynomials $f = f_0 + f_1x + f_2x^2 + f_3x^3$ and $g = g_0 + g_1x + g_2x^2 + g_3x^3$:

$$\begin{aligned}
r_0 &= f_0g_0\\
r_1 &= f_0g_1 + f_1g_0\\
r_2 &= f_0g_2 + f_1g_1 + f_2g_0\\
r_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0\\
r_4 &= f_1g_3 + f_2g_2 + f_3g_1\\
r_5 &= f_2g_3 + f_3g_2\\
r_6 &= f_3g_3
\end{aligned}$$

- Ignore carries, overflows etc. for a moment
- 16 multiplications, 9 additions
- How to vectorize multiplications?

Vectorization problems III

Data shuffeling

$$
\begin{aligned}
r_0 &= f_0g_0\\
r_1 &= f_0g_1 + f_1g_0\\
r_2 &= f_0g_2 + f_1g_1 + f_2g_0\\
r_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0\\
r_4 &= f_1g_3 + f_2g_2 + f_3g_1\\
r_5 &= f_2g_3 + f_3g_2\\
r_6 &= f_3g_3
\end{aligned}
$$

- Can easily load (f_0, f_1, f_2, f_3) and (g_0, g_1, g_2, g_3)
- Multiply, obtain $(f_0g_0, f_1g_1, f_2g_2, f_3g_3)$
- And now what?
- Answer: Need to *shuffle* data in input and output registers
- Significant overhead, not clear that vectorization speeds up computation!

Efficient vectorization

- Most important question: Where does the parallelism come from?
- Easiest answer: Consider multiple batched encryptions, decryptions, signature computations, verifications, etc. (but that increases latency)
- Often: Can exploit lower-level parallelism
- Rule of thumb: parallelize on an as high as possible level
- Vectorization is hard to do as “add-on” optimization
- Reconsider algorithms and data structures, synergy with constant-time algorithms

Bitslicing

- Imagine registers that have only one bit
- Perform arithmetic on those registers using `XOR`, `AND`, `OR`
- Essentially the same as hardware implementations
- But wait, registers are longer!
- Think of them as vectors of bits
- This needs transposition of the "binary data matrix"
- Perform the simulated hardware implementations on many independent data streams
- Bitslicing works for every algorithm
- Bitslicing is inherently protected against timing attacks
- Efficient bitslicing needs a huge amount of data-level parallelism

Bitslicing binary polynomials

4-coefficient binary polynomials

$(a_3x^3 + a_2x^2 + a_1x + a_0)$, with $a_i \in \{0,1\}$

4-coefficient bitsliced binary polynomials

```
typedef unsigned char poly4; /* 4 coefficients in the low 4 bits */
typedef unsigned long long poly4x64[4];

void poly4_bitslice(poly4x64 r, const poly4 x[64])
{
  int i,j;
  for(i=0;i<4;i++)
  {
    r[i] = 0;
    for(j=0;j<64;j++)
      r[i] |= (unsigned long long)(1 & (x[j] >> i))<<j;
  }
}
```

Bitsliced binary-polynomial multiplication

```
typedef unsigned long long poly4x64[4];
typedef unsigned long long poly7x64[7];

void poly4x64_mul(poly7x64 r, const poly4x64 a, const poly4x64 b)
{
  r[0] = a[0] & b[0];
  r[1] = (a[0] & b[1]) ^ (a[1] & b[0]);
  r[2] = (a[0] & b[2]) ^ (a[1] & b[1]) ^ (a[2] & b[0]);
  r[3] = (a[0] & b[3]) ^ (a[1] & b[2]) ^ (a[2] & b[1]) ^ (a[3] & b[0]);
  r[4] = (a[1] & b[3]) ^ (a[2] & b[2]) ^ (a[3] & b[1]);
  r[5] = (a[2] & b[3]) ^ (a[3] & b[2]);
  r[6] = (a[3] & b[3]);
}
```

Bitslicing issues

- XOR, AND, OR, etc are usually fast (e.g., 3 128-bit operations per cycle on Intel Core 2)
- Can be very fast for operations that are not natively supported (like arithmetic in binary fields)
- Active data set increases massively (e.g., $128\times$)
- For "normal" vector operations, register space is increased accordingly (e.g, 16 256-bit vector registers vs. 16 64-bit integer registers)
- For bitslicing: Need to fit more data into the same registers
- Typical consequence: more loads and stores (that easily become the performance bottleneck)