High-Performance Cryptography in Software

Peter Schwabe
Research Center for Information Technology Innovation
Academia Sinica

September 3, 2012

ECRYPT Summer School: Challenges in Security Engineering 2012
Introduction

- Previous talk: High performance crypto in hardware
- Reason for special-purpose crypto hardware: Speed!
- Disadvantages: High cost, loss of flexibility, hard to replace/update
Introduction

- Previous talk: High performance crypto in hardware
- Reason for special-purpose crypto hardware: Speed!
- Disadvantages: High cost, loss of flexibility, hard to replace/update
- This talk: How fast can we make crypto on off-the-shelf computers?
- Implement cryptography with a set of general-purpose instructions
Levels of optimization

- Consider the example of elliptic-curve cryptography
- Various levels of optimization:
  - Choice of scalar-multiplication algorithm
  - Choice of curve and underlying finite field
  - Choice of coordinates and addition and doubling formulas
  - Representation of finite-field elements in machine words and related algorithms (e.g. schoolbook vs. Karatsuba multiplication)
  - Low-level optimizations of machine instructions
Levels of optimization

- Consider the example of elliptic-curve cryptography
- Various levels of optimization:
  - Choice of scalar-multiplication algorithm
  - Choice of curve and underlying finite field
  - Choice of coordinates and addition and doubling formulas
  - Representation of finite-field elements in machine words and related algorithms (e.g. schoolbook vs. Karatsuba multiplication)
  - Low-level optimizations of machine instructions
- These levels are not independent, many subtle interactions
Consider the example of elliptic-curve cryptography

Various levels of optimization:

- Choice of scalar-multiplication algorithm
- Choice of curve and underlying finite field
- Choice of coordinates and addition and doubling formulas
- Representation of finite-field elements in machine words and related algorithms (e.g. schoolbook vs. Karatsuba multiplication)
- Low-level optimizations of machine instructions

These levels are not independent, many subtle interactions
A program is a sequence of instructions.

Load/Store instructions move data between memory and registers (processed by the L/S unit).

Branch instructions (conditionally) jump to a position in the program.

Arithmetic instructions perform simple operations on values in registers (processed by the ALU).

Registers are fast (fixed-size) storage units, addressed “by name.”
A first program
Adding up 1000 integers

1. Set register R1 to zero
2. Set register R2 to zero
3. Load 32-bits from address START+R2 into register R3
4. Add 32-bit integers in R1 and R3, write the result in R1
5. Increase value in register R2 by 4
6. Compare value in register R2 to 4000
7. Goto line 3 if R2 was smaller than 4000
A first program
Adding up 1000 integers in readable syntax

```c
int32 result
int32 tmp
int32 ctr

result = 0
ctr = 0
looptop:
    tmp = mem32[START+ctr]
    result += tmp
    ctr += 4
    unsigned<? ctr - 4000
    goto looptop if unsigned<
```
Running the program

- Easy approach: Per “time-slot” (cycle) execute one instruction, then go for the next
- Cycles needs to be long enough to finish the most complex supported instruction

- Overlap instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)
- This is called pipelined execution (many more stages possible)
- Advantage: cycles can be much shorter (higher clock speed)
- Requirement for overlapping execution: instructions have to be independent
Running the program

- Easy approach: Per “time-slot” (*cycle*) execute one instruction, then go for the next
- Cycles needs to be long enough to finish the most complex supported instruction
- Other approach: Chop instructions into smaller tasks, e.g. for addition:
  1. Fetch instruction
  2. Decode instruction
  3. Fetch register arguments
  4. Execute (actual addition)
  5. Write back to register
Running the program

▶ Easy approach: Per “time-slot” (cycle) execute one instruction, then go for the next

▶ Cycles needs to be long enough to finish the most complex supported instruction

▶ Other approach: Chop instructions into smaller tasks, e.g. for addition:
  1. Fetch instruction
  2. Decode instruction
  3. Fetch register arguments
  4. Execute (actual addition)
  5. Write back to register

▶ Overlap instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)

▶ This is called pipelined execution (many more stages possible)

▶ Advantage: cycles can be much shorter (higher clock speed)
Running the program

▶ Easy approach: Per “time-slot” (*cycle*) execute one instruction, then go for the next
▶ Cycles needs to be long enough to finish the most complex supported instruction
▶ Other approach: Chop instructions into smaller tasks, e.g. for addition:
   1. Fetch instruction
   2. Decode instruction
   3. Fetch register arguments
   4. Execute (actual addition)
   5. Write back to register
▶ Overlap instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)
▶ This is called pipelined execution (many more stages possible)
▶ Advantage: cycles can be much shorter (higher *clock speed*)
▶ Requirement for overlapping execution: instructions have to be independent
Throughput and latency

- While the ALU is executing an instruction the L/S and branch units are idle
Throughput and latency

- While the ALU is executing an instruction the L/S and branch units are idle
- Idea: Duplicate fetch and decode, handle two or three instructions per cycle
- While we’re at it: Why not deploy two ALUs
- This concept is called *superscalar* execution
Throughput and latency

- While the ALU is executing an instruction the L/S and branch units are idle
- Idea: Duplicate fetch and decode, handle two or three instructions per cycle
- While we’re at it: Why not deploy two ALUs
- This concept is called *superscalar* execution
- Number of independent instructions of one type per cycle: **throughput**
- Number of cycles that need to pass before the result can be used: **latency**
An example computer
Still highly simplified

Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle
Adding up 1000 integers on this computer

- Need at least 1000 load instructions: \( \geq 1000 \) cycles

Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle
Adding up 1000 integers on this computer

- Need at least 1000 load instructions: $\geq 1000$ cycles
- Need at least 999 addition instructions: $\geq 500$ cycles

Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle
Adding up 1000 integers on this computer

- Need at least 1000 load instructions: $\geq 1000$ cycles
- Need at least 999 addition instructions: $\geq 500$ cycles
- At least 1999 instructions: $\geq 500$ cycles

Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle
Adding up 1000 integers on this computer

- Need at least 1000 load instructions: $\geq 1000$ cycles
- Need at least 999 addition instructions: $\geq 500$ cycles
- At least 1999 instructions: $\geq 500$ cycles
- **Lower bound**: 1000 cycles

### Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle
How about our program?

```c
int32 result
int32 tmp
int32 ctr

result = 0
ctr = 0
looptop:
    tmp = mem32[START+ctr]
    result += tmp
    ctr += 4
    unsigned<? ctr - 4000
    goto looptop if unsigned<
```
How about our program?

```c
int32 result
int32 tmp
int32 ctr

result = 0
ctr = 0
looptop:
    tmp = mem32[START+ctr]
    # wait 2 cycles for tmp
    result += tmp
    ctr += 4
    # wait 1 cycle for ctr
    unsigned<? ctr - 4000
    # wait 1 cycle for unsigned<
goto looptop if unsigned<
```

- Addition has to wait for load
- Comparison has to wait for addition
- Each iteration of the loop takes 8 cycles
- Total: > 8000 cycles
How about our program?

```c
int32 result
int32 tmp
int32 ctr

result = 0
ctr    = 0
looptop:
    tmp = mem32[START+ctr]
    # wait 2 cycles for tmp
    result += tmp
    ctr += 4
    # wait 1 cycle for ctr
    unsigned<? ctr - 4000
    # wait 1 cycle for unsigned<
goto looptop if unsigned<
```

- Addition has to wait for load
- Comparison has to wait for addition
- Each iteration of the loop takes 8 cycles
- Total: \( > 8000 \) cycles
- This program sucks!
Making the program fast
Step 1 – Unrolling

result = 0
tmp = mem32[START+0]
result += tmp
tmp = mem32[START+4]
result += tmp
tmp = mem32[START+8]
result += tmp
...
tmp = mem32[START+3996]
result += tmp

▶ Remove all the loop control: unrolling
Making the program fast
Step 1 – Unrolling

```
result = 0
tmp = mem32[START+0]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+4]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+8]
# wait 2 cycles for tmp
result += tmp
...

tmp = mem32[START+3996]  
# wait 2 cycles for tmp
result += tmp
```

- Remove all the loop control: *unrolling*
- Each load-and-add now takes 3 cycles
- Total: $\approx 3000$ cycles
Making the program fast
Step 1 – Unrolling

result = 0
tmp = mem32[START+0]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+4]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+8]
# wait 2 cycles for tmp
result += tmp
...
tmp = mem32[START+3996]
# wait 2 cycles for tmp
result += tmp

- Remove all the loop control: unrolling
- Each load-and-add now takes 3 cycles
- Total: \( \approx 3000 \) cycles
- Better, but still too slow
Making the program fast
Step 2 – Instruction Scheduling

```c
result = mem32[START + 0]
tmp0 = mem32[START + 4]
tmp1 = mem32[START + 8]
tmp2 = mem32[START + 12]
result += tmp0
result += tmp1
result += tmp2
tmp2 = mem32[START + 16]
result += tmp0
result += tmp1
result += tmp2
...```

- Load values earlier
- Load latencies are hidden
- Use more registers for loaded values (tmp0, tmp1, tmp2)
- Get rid of one addition to zero
Making the program fast

Step 2 – Instruction Scheduling

```c
result = mem32[START + 0]
tmp0  = mem32[START + 4]
tmp1  = mem32[START + 8]
tmp2  = mem32[START +12]
result += tmp0
tmp0  = mem32[START+16]
# wait 1 cycle for result
result += tmp1
tmp1  = mem32[START+20]
# wait 1 cycle for result
result += tmp2
tmp2  = mem32[START+24]
...
result += tmp2
tmp2  = mem32[START+3996]
# wait 1 cycle for result
result += tmp0
# wait 1 cycle for result
result += tmp1
# wait 1 cycle for result
result += tmp2
```

- Load values earlier
- Load latencies are hidden
- Use more registers for loaded values (tmp0, tmp1, tmp2)
- Get rid of one addition to zero
- Now arithmetic latencies kick in
- Total: \( \approx 2000 \) cycles
Making the program fast
Step 3 – More Instruction Scheduling (two accumulators)

```
result0 = mem32[START + 0]
tmp0 = mem32[START + 8]
result1 = mem32[START + 4]
tmp1 = mem32[START +12]
tmp2 = mem32[START +16]

result0 += tmp0
result0 += tmp1
result1 += tmp2
result0 += tmp0
result0 += tmp1
```

- Use one more accumulator register (result1)
- All latencies hidden
- Total: 1004 cycles
- Asymptotically \( n \) cycles for \( n \) additions
Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop

Note: Good instruction scheduling typically requires more registers
Opposing requirements to register allocation (assigning registers to live variables, minimizing memory access)
Both instruction scheduling and register allocation are NP hard
So is the joint problem
Many instances are efficiently solvable
Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (instruction scheduling)

Resulting program is larger and requires more registers!

Note: Good instruction scheduling typically requires more registers

Opposing requirements to register allocation (assigning registers to live variables, minimizing memory access)

Both instruction scheduling and register allocation are NP hard

So is the joint problem

Many instances are efficiently solvable
Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (instruction scheduling)
  - Resulting program is larger and requires more registers!

Note: Good instruction scheduling typically requires more registers
Opposing requirements to register allocation (assigning registers to live variables, minimizing memory access)
Both instruction scheduling and register allocation are NP hard
So is the joint problem
Many instances are efficiently solvable
Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (instruction scheduling)
    - Resulting program is larger and requires more registers!
- Note: Good instruction scheduling typically requires more registers
Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (instruction scheduling)
  - Resulting program is larger and requires more registers!
- Note: Good instruction scheduling typically requires more registers
- Opposing requirements to register allocation (assigning registers to live variables, minimizing memory access)
Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (instruction scheduling)
    - Resulting program is larger and requires more registers!
- Note: Good instruction scheduling typically requires more registers
- Opposing requirements to register allocation (assigning registers to live variables, minimizing memory access)
- Both instruction scheduling and register allocation are NP hard
- So is the joint problem
- Many instances are efficiently solvable
What instructions and how many registers do we have?

- Instructions are defined by the **instruction set**
- Supported register names are defined by the **set of architectural registers**
- Instruction set and set of architectural registers together define the **architecture**
- Examples for architectures: x86, AMD64, ARMv6, ARMv7, UltraSPARC
- Sometimes base architectures are extended, e.g., MMX, SSE, NEON
Architectures and microarchitectures

What instructions and how many registers do we have?

- Instructions are defined by the **instruction set**
- Supported register names are defined by the **set of architectural registers**
- Instruction set and set of architectural registers together define the **architecture**
- Examples for architectures: x86, AMD64, ARMv6, ARMv7, UltraSPARC
- Sometimes base architectures are extended, e.g., MMX, SSE, NEON

What determines latencies etc?

- Different **microarchitectures** implement an architecture
- Latencies and throughputs are specific to a microarchitecture
- Example: Intel Core 2 Quad Q9550 implements the AMD64 architecture
Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture.
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture.
- Many software is shipped in binary form (cannot recompile).
Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)
- Idea: Let the processor reschedule instructions on the fly
- Look ahead a few instructions, pick one that can be executed
- This is called **out-of-order execution**
Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)
- Idea: Let the processor reschedule instructions on the fly
- Look ahead a few instructions, pick one that can be executed
- This is called out-of-order execution
- Typically requires more physical than architectural registers and register renaming
Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)
- Idea: Let the processor reschedule instructions on the fly
- Look ahead a few instructions, pick one that can be executed
- This is called **out-of-order execution**
- Typically requires more physical than architectural registers and **register renaming**
- Harder for the (assembly) programmer to understand what exactly will happen with the code
- Harder to come up with optimal scheduling
Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)
- Idea: Let the processor reschedule instructions on the fly
- Look ahead a few instructions, pick one that can be executed
- This is called **out-of-order execution**
- Typically requires more physical than architectural registers and **register renaming**
- Harder for the (assembly) programmer to understand what exactly will happen with the code
- Harder to come up with optimal scheduling
- Harder to screw up completely
The Advanced Encryption Standard (AES)

- Block cipher Rijndael proposed by Rijmen, Daemen in 1998
- Selected as AES by NIST in October 2000
The Advanced Encryption Standard (AES)

- Block cipher Rijndael proposed by Rijmen, Daemen in 1998
- Selected as AES by NIST in October 2000
- Block size: 128 bits (AES state: 4x4 matrix of 16 bytes)
- Key size 128/192/256 bits (resp. 10/12/14 rounds)
The Advanced Encryption Standard (AES) ▶

- Block cipher Rijndael proposed by Rijmen, Daemen in 1998
- Selected as AES by NIST in October 2000
- Block size: 128 bits (AES state: 4x4 matrix of 16 bytes)
- Key size 128/192/256 bits (resp. 10/12/14 rounds)
The Advanced Encryption Standard (AES)

- Block cipher Rijndael proposed by Rijmen, Daemen in 1998
- Selected as AES by NIST in October 2000
- Block size: 128 bits (AES state: 4x4 matrix of 16 bytes)
- Key size 128/192/256 bits (resp. 10/12/14 rounds)
- AES with $n$ rounds uses $n + 1$ 16-byte rounds keys $K_0, \ldots, K_n$
The Advanced Encryption Standard (AES) ▶ Block cipher Rijndael proposed by Rijmen, Daemen in 1998 ▶ Selected as AES by NIST in October 2000 ▶ Block size: 128 bits (AES state: 4x4 matrix of 16 bytes) ▶ Key size 128/192/256 bits (resp. 10/12/14 rounds) ▶ AES with \( n \) rounds uses \( n + 1 \) 16-byte rounds keys \( K_0, \ldots, K_n \) ▶ Four operations per round: SubBytes, ShiftRows, MixColumns, and AddRoundKey ▶ Last round does not have MixColumns
High-level pseudocode AES-128

Require: 128-bit input block $B$, 128-bit AES round keys $K_0, \ldots, K_{10}$
Ensure: 128-bit block of encrypted output

\[
B \leftarrow \text{AddRoundKey}(B, K_0)
\]

for $i$ from 1 to 9 do
  \[
  B \leftarrow \text{SubBytes}(B)
  \]
  \[
  B \leftarrow \text{ShiftRows}(B)
  \]
  \[
  B \leftarrow \text{MixColumns}(B)
  \]
  \[
  B \leftarrow \text{AddRoundKey}(B, K_i)
  \]
end for

\[
B \leftarrow \text{SubBytes}(B)
\]
\[
B \leftarrow \text{ShiftRows}(B)
\]
\[
B \leftarrow \text{AddRoundKey}(B, K_{10})
\]
return $B$
The AES operations, part I

- SubBytes is an S-Box acting on individual bytes
- Substitution based on inversion in $\mathbb{F}_{2^8}$

![SubBytes S-Box diagram]

- ShiftRows rotates each row by a different amount

![ShiftRows rotation diagram]
The AES operations, part II

- **MixColumns** is a linear transformation on columns

\[
\begin{array}{cccc}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33} \\
\end{array}
\]

\[
\begin{array}{cccc}
  b_{00} & b_{01} & b_{02} & b_{03} \\
  b_{10} & b_{11} & b_{12} & b_{13} \\
  b_{20} & b_{21} & b_{22} & b_{23} \\
  b_{30} & b_{31} & b_{32} & b_{33} \\
\end{array}
\]

- **AddRoundKey** XORs the 128-bit round key to the state

\[
\begin{array}{cccc}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33} \\
\end{array}
\]

\[
\begin{array}{cccc}
  k_{r00} & k_{r01} & k_{r02} & k_{r03} \\
  k_{r10} & k_{r11} & k_{r12} & k_{r13} \\
  k_{r20} & k_{r21} & k_{r22} & k_{r23} \\
  k_{r30} & k_{r31} & k_{r32} & k_{r33} \\
\end{array}
\]
AES on 32-bit processors

- Idea from the AES proposal: Merge SubBytes, ShiftRows, and MixColumns
- Use 4 lookup tables $T_0$, $T_1$, $T_2$, and $T_3$ (1 KB each)
AES on 32-bit processors

- Idea from the AES proposal: Merge SubBytes, ShiftRows, and MixColumns
- Use 4 lookup tables $T0$, $T1$, $T2$, and $T3$ (1 KB each)

The first round of AES in C

- Input: 32-bit integers $y0$, $y1$, $y2$, $y3$
- Output: 32-bit integers $z0$, $z1$, $z2$, $z3$
- Round keys in 32-bit-integer array $rk[44]$

```c
z0 = T0[ y0 >> 24 ] ^ T1[ (y1 >> 16) & 0xff ] ^ T2[ (y2 >> 8) & 0xff ] ^ T3[ y3 & 0xff ] ^ rk[4];
z1 = T0[ y1 >> 24 ] ^ T1[ (y2 >> 16) & 0xff ] ^ T2[ (y3 >> 8) & 0xff ] ^ T3[ y0 & 0xff ] ^ rk[5];
z2 = T0[ y2 >> 24 ] ^ T1[ (y3 >> 16) & 0xff ] ^ T2[ (y0 >> 8) & 0xff ] ^ T3[ y1 & 0xff ] ^ rk[6];
z3 = T0[ y3 >> 24 ] ^ T1[ (y0 >> 16) & 0xff ] ^ T2[ (y1 >> 8) & 0xff ] ^ T3[ y2 & 0xff ] ^ rk[7];
```
unsigned char rk[176], T0[1024], T1[1024], T2[1024], T3[1024];

z0 = *(uint32 *)(rk + 16);
z1 = *(uint32 *)(rk + 20);
z2 = *(uint32 *)(rk + 24);
z3 = *(uint32 *)(rk + 28);

z0 ^= *(uint32 *) (T0 + ((y0 >> 22) & 0x3fc)) ^ *(uint32 *) (T1 + ((y1 >> 14) & 0x3fc)) ^ *(uint32 *) (T2 + ((y2 >> 6) & 0x3fc)) ^ *(uint32 *) (T3 + ((y3 << 2) & 0x3fc));
z1 ^= *(uint32 *) (T0 + ((y1 >> 22) & 0x3fc)) ^ *(uint32 *) (T1 + ((y2 >> 14) & 0x3fc)) ^ *(uint32 *) (T2 + ((y3 >> 6) & 0x3fc)) ^ *(uint32 *) (T3 + ((y0 << 2) & 0x3fc));
z2 ^= *(uint32 *) (T0 + ((y2 >> 22) & 0x3fc)) ^ *(uint32 *) (T1 + ((y3 >> 14) & 0x3fc)) ^ *(uint32 *) (T2 + ((y0 >> 6) & 0x3fc)) ^ *(uint32 *) (T3 + ((y1 << 2) & 0x3fc));
z3 ^= *(uint32 *) (T0 + ((y3 >> 22) & 0x3fc)) ^ *(uint32 *) (T1 + ((y0 >> 14) & 0x3fc)) ^ *(uint32 *) (T2 + ((y1 >> 6) & 0x3fc)) ^ *(uint32 *) (T3 + ((y2 << 2) & 0x3fc));
AES instruction counts

- Each round has 20 loads, 16 shifts, 16 masks and 16 xors
- Last round is slightly different: Needs 16 more mask instructions
- 4 load instructions to load input, 4 stores for output
- In CTR mode: 4 xors with the key stream, incrementing the counter
- Results in 720 instructions needed to encrypt a block of 16 bytes
- Specifically: 208 loads, 4 stores, 508 arithmetic instructions
AES instruction counts

- Each round has 20 loads, 16 shifts, 16 masks and 16 xors
- Last round is slightly different: Needs 16 more mask instructions
- 4 load instructions to load input, 4 stores for output
Each round has 20 loads, 16 shifts, 16 masks and 16 xors

Last round is slightly different: Needs 16 more mask instructions

4 load instructions to load input, 4 stores for output

In CTR mode: 4 xors with the key stream, incrementing the counter

...some more overhead

Results in 720 instructions needed to encrypt a block of 16 bytes

Specifically: 208 loads, 4 stores, 508 arithmetic instructions
Making AES fast on an UltraSPARC
My first project as Ph.D. student

- 64-bit architecture
- Up to 4 instructions per cycle
- At most 2 integer-arithmetic instructions per cycle
- At most 1 load/store instruction per cycle
- 24 integer registers available
Making AES fast on an UltraSPARC
My first project as Ph.D. student

- 64-bit architecture
- Up to 4 instructions per cycle
- At most 2 integer-arithmetic instructions per cycle
- At most 1 load/store instruction per cycle
- 24 integer registers available
- Previous AES speed:
  - 20.75 cycles/byte by Bernstein (public domain)
Making AES fast on an UltraSPARC
My first project as Ph.D. student

- 64-bit architecture
- Up to 4 instructions per cycle
- At most 2 integer-arithmetic instructions per cycle
- At most 1 load/store instruction per cycle
- 24 integer registers available
- Previous AES speed:
  - 20.75 cycles/byte by Bernstein (public domain)
  - 16.875 cycles/byte by Lipmaa (unpublished)
Making AES fast on an UltraSPARC
My first project as Ph.D. student

Computing a lower bound

Reminder: 208 loads, 4 stores, 508 integer instructions per 16-byte block

- Only one load or store per cycle ($\Rightarrow$ at least 212 cycles)
- Only 2 arithmetic instructions per cycle ($\Rightarrow$ at least 254 cycles)

Dan’s reaction:
"...this is no time to relax; you have to not just beat Lipmaa’s code, but beat it to a bloody pulp and dance on its grave. :-)"

After writing a simplified simulator and more instruction scheduling: 254 cycles/block, 15.98 cycles/byte

What now? Is this already a bloody pulp?
Making AES fast on an UltraSPARC
My first project as Ph.D. student

Computing a lower bound
Reminder: 208 loads, 4 stores, 508 integer instructions per 16-byte block
  ▶ Only one load or store per cycle ($\Rightarrow$ at least 212 cycles)
  ▶ Only 2 arithmetic instructions per cycle ($\Rightarrow$ at least 254 cycles)

Making it fast
  ▶ After quite some instruction scheduling: 269 cycles per block
Making AES fast on an UltraSPARC
My first project as Ph.D. student

Computing a lower bound
Reminder: 208 loads, 4 stores, 508 integer instructions per 16-byte block
- Only one load or store per cycle ($\Rightarrow$ at least 212 cycles)
- Only 2 arithmetic instructions per cycle ($\Rightarrow$ at least 254 cycles)

Making it fast
- After quite some instruction scheduling: 269 cycles per block
- Dan’s reaction:
  “… this is no time to relax; you have to not just beat Lipmaa’s code, but beat it to a bloody pulp and dance on its grave. :-)”
Making AES fast on an UltraSPARC
My first project as Ph.D. student

Computing a lower bound
Reminder: 208 loads, 4 stores, 508 integer instructions per 16-byte block
- Only one load or store per cycle (⇒ at least 212 cycles)
- Only 2 arithmetic instructions per cycle (⇒ at least 254 cycles)

Making it fast
- After quite some instruction scheduling: 269 cycles per block
- Dan’s reaction:
  “...this is no time to relax; you have to not just beat Lipmaa’s code, but beat it to a bloody pulp and dance on its grave. :-)”
- After writing a simplified simulator and more instruction scheduling: 254 cycles/block, 15.98 cycles/byte
Computing a lower bound

Reminder: 208 loads, 4 stores, 508 integer instructions per 16-byte block
- Only one load or store per cycle (⇒ at least 212 cycles)
- Only 2 arithmetic instructions per cycle (⇒ at least 254 cycles)

Making it fast

- After quite some instruction scheduling: 269 cycles per block
- Dan’s reaction:
  “… this is no time to relax; you have to not just beat Lipmaa’s code, but beat it to a bloody pulp and dance on its grave. :-)”

- After writing a simplified simulator and more instruction scheduling: 254 cycles/block, 15.98 cycles/byte
- What now? Is this already a bloody pulp?
Making AES fast on an UltraSPARC

Lowering the lower bound

- We have to reduce the number of (arithmetic) instructions
- Idea: The UltraSPARC is a 64-bit architecture, pad 32-bit values with zeros, i.e.,
  \[0xc66363a5 \text{ becomes } 0x0c60063006300a50\]
- Do that consistently for values in registers, the tables and the round keys
- Interleave entries in tables \(T0\) and \(T1\) and in \(T2\) and \(T3\)

\begin{verbatim}
Without padded registers
  t0 = (uint32) y0 >> 22
  t1 = (uint32) y0 >> 14
  t2 = (uint32) y0 >> 6
  t3 = (uint32) y0 << 2
  t0 &= 0x7f8
  t1 &= 0x7f8
  t2 &= 0x7f8
  t3 &= 0x7f8

With padded registers
\end{verbatim}
Making AES fast on an UltraSPARC

Lowering the lower bound

- We have to reduce the number of (arithmetic) instructions
- Idea: The UltraSPARC is a 64-bit architecture, pad 32-bit values with zeros, i.e.,
  0xc66363a5 becomes 0x0c60063006300a50
- Do that consistently for values in registers, the tables and the round keys
- Interleave entries in tables T0 and T1 and in T2 and T3

### Without padded registers

```c
  t0 = (uint32) y0 >> 22
  t1 = (uint32) y0 >> 14
  t2 = (uint32) y0 >> 6
  t3 = (uint32) y0 << 2
  t0 &= 0x7f8
  t1 &= 0x7f8
  t2 &= 0x7f8
  t3 &= 0x7f8
```

### With padded registers

```c
  t0 = (uint64) y0 >> 48
  t1 = (uint64) y0 >> 32
  t2 = (uint64) y0 >> 16
  t1 &= 0xff0
  t2 &= 0xff0
  t3 = y0 & 0xff0
```
Making AES fast on an UltraSPARC

Lowering the lower bound

- We have to reduce the number of (arithmetic) instructions
- Idea: The UltraSPARC is a 64-bit architecture, pad 32-bit values with zeros, i.e., 0xc66363a5 becomes 0x0c600606060a50
- Do that consistently for values in registers, the tables and the round keys
- Interleave entries in tables T0 and T1 and in T2 and T3
- Instruction set supports 32-bit shifts that zero out the upper 32 bits

Without padded registers

\[
\begin{align*}
  t0 &= (\text{uint32}) \ y0 \gg 22 \\
  t1 &= (\text{uint32}) \ y0 \gg 14 \\
  t2 &= (\text{uint32}) \ y0 \gg 6 \\
  t3 &= (\text{uint32}) \ y0 \ll 2 \\
  t0 &= 0x7f8 \\
  t1 &= 0x7f8 \\
  t2 &= 0x7f8 \\
  t3 &= 0x7f8
\end{align*}
\]

With padded registers

\[
\begin{align*}
  t0 &= (\text{uint64}) \ y0 \gg 48 \\
  t1 &= (\text{uint64}) \ y0 \gg 32 \\
  t2 &= (\text{uint32}) \ y0 \gg 16 \\
  t1 &= 0xff0 \\
  t3 &= y0 \& 0xff0
\end{align*}
\]
Making AES fast on an UltraSPARC

Lowering the lower bound

- We have to reduce the number of (arithmetic) instructions
- Idea: The UltraSPARC is a 64-bit architecture, pad 32-bit values with zeros, i.e.,
  0xc66363a5 becomes 0x0c6006006300a50
- Do that consistently for values in registers, the tables and the round keys
- Interleave entries in tables T0 and T1 and in T2 and T3
- Instruction set supports 32-bit shifts that zero out the upper 32 bits
- Apply some more optimizations
- Final result: AES in CTR mode on UltraSPARC III at 12.06 cycles/byte
So far there was nothing crypto-specific in this talk (except for the AES example)

Is optimizing crypto the same as optimizing any other software?
So far there was nothing crypto-specific in this talk (except for the AES example)

Is optimizing crypto the same as optimizing any other software?

No.
So far there was nothing crypto-specific in this talk (except for the AES example).

Is optimizing crypto the same as optimizing any other software?

No. Cryptographic software deals with secret data (keys).

Information about secret data must not leak.
Optimizing crypto vs. optimizing ... 

- So far there was nothing crypto-specific in this talk (except for the AES example)
- Is optimizing crypto the same as optimizing any other software?
- No. Cryptographic software deals with secret data (keys)
- Information about secret data must not leak
- Most critical for software implementations on “large” CPUs: software must take constant time (independent of secret data)
Optimizing crypto vs. optimizing . . .

- So far there was nothing crypto-specific in this talk (except for the AES example)
- Is optimizing crypto the same as optimizing any other software?
- No. Cryptographic software deals with secret data (keys)
- Information about secret data must not leak
- Most critical for software implementations on “large” CPUs: software must take constant time (independent of secret data)
- Is this the case for the AES implementation?
Cached memory access

- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
Cached memory access

- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
- A load from memory places data also in the cache
- Data remains in cache until it’s replaced by other data
Cached memory access

- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
- A load from memory places data also in the cache
- Data remains in cache until it’s replaced by other data
- Loading data is fast if data is in the cache (**cache hit**)
- Loading data is slow if data is not in the cache (**cache miss**)

Diagram:

- Branch Unit
- ALU
- Registers
- L/S Unit
- Cache
- CPU
- Memory

Arrows indicate data flow and relationships between components.
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache

<table>
<thead>
<tr>
<th>$T0[0]$ ... $T0[15]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T0[16]$ ... $T0[31]$</td>
</tr>
<tr>
<td>$T0[32]$ ... $T0[47]$</td>
</tr>
<tr>
<td>$T0[48]$ ... $T0[63]$</td>
</tr>
<tr>
<td>$T0[64]$ ... $T0[79]$</td>
</tr>
<tr>
<td>$T0[80]$ ... $T0[95]$</td>
</tr>
<tr>
<td>$T0[96]$ ... $T0[111]$</td>
</tr>
<tr>
<td>$T0[112]$ ... $T0[127]$</td>
</tr>
<tr>
<td>$T0[128]$ ... $T0[143]$</td>
</tr>
<tr>
<td>$T0[144]$ ... $T0[159]$</td>
</tr>
<tr>
<td>$T0[160]$ ... $T0[175]$</td>
</tr>
<tr>
<td>$T0[176]$ ... $T0[191]$</td>
</tr>
<tr>
<td>$T0[192]$ ... $T0[207]$</td>
</tr>
<tr>
<td>$T0[208]$ ... $T0[223]$</td>
</tr>
<tr>
<td>$T0[224]$ ... $T0[239]$</td>
</tr>
<tr>
<td>$T0[240]$ ... $T0[255]$</td>
</tr>
</tbody>
</table>
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
- AES continues, loads from table again
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
- AES continues, loads from table again
- Attacker loads his data:
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
- AES continues, loads from table again
- Attacker loads his data:
  - Fast: cache hit (AES did not just load from this line)

High-Performance Cryptography in Software
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
- AES continues, loads from table again
- Attacker loads his data:
  - Fast: cache hit (AES did not just load from this line)
  - Slow: cache miss (AES just loaded from this line)
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
- AES continues, loads from table again
- Attacker loads his data:
  - Fast: cache hit (AES did not just load from this line)
  - Slow: cache miss (AES just loaded from this line)
- Cache-timing attack by Osvik, Tromer, Shamir from 2006: 65 ms to steal a 256-bit AES key used for Linux hard-disk encryption
More timing attacks

- **Bad news:** Loading from secret positions is not the only source for timing variation
More timing attacks

- **Bad news:** Loading from secret positions is not the only source for timing variation
- More obvious: Secret branch conditions:
  ```
  if s then
      do A
  else
      do B
  end if
  ```
- **Good news:** Loads from secret indices and secret branch conditions are the only problems (on most processors)
More timing attacks

- **Bad news:** Loading from secret positions is not the only source for timing variation
- More obvious: Secret branch conditions:
  
  ```
  if s then
      do A
  else
      do B
  end if
  ```
- Even if A and B take the same amount of cycles this is *not* constant time!
- Reason: Conditional branch takes different amount of cycles whether taken or not
More timing attacks

- **Bad news:** Loading from secret positions is not the only source for timing variation
- More obvious: Secret branch conditions:
  ```c
  if s then
    do A
  else
    do B
  end if
  ```
  Even if A and B take the same amount of cycles this is *not* constant time!
- Reason: Conditional branch takes different amount of cycles whether taken or not

- **Good news:** Loads from secret indices and secret branch conditions are the only problems (on most processors)
Eliminating ifs and lookups

Generic technique to eliminate conditional branches

\[
\text{if } s \text{ then } \quad a \leftarrow b \\
\text{else} \quad \quad a \leftarrow c \\
\text{end if}
\]

\[
a \leftarrow s \cdot b + (1 - s) \cdot c
\]
Eliminating ifs and lookups

Generic technique to eliminate conditional branches

```plaintext
if \ s \ then
  a \leftarrow \ b
else
  a \leftarrow \ c
end if
```

\[
a \leftarrow s \cdot b + (1 - s) \cdot c
\]

- If computation of \( b \) and \( c \) is cheap, this may even speed up the code
Eliminating ifs and lookups

Generic technique to eliminate conditional branches

\[
\begin{align*}
\text{if } s & \text{ then } \\
    a & \leftarrow b \\
\text{else } & \\
    a & \leftarrow c \\
\text{end if}
\end{align*}
\]

\[a \leftarrow s \cdot b + (1 - s) \cdot c\]

If computation of \( b \) and \( c \) is cheap, this may even speed up the code

Generic technique to eliminate lookups

- Load all possible values from the table
- Use arithmetic (similar as for elimination of conditional branches) to pick the right one
Eliminating ifs and lookups

Generic technique to eliminate conditional branches

if \( s \) then
  \( a \leftarrow b \)
else
  \( a \leftarrow c \)
end if

\[ a \leftarrow s \cdot b + (1 - s) \cdot c \]

- If computation of \( b \) and \( c \) is cheap, this may even speed up the code

Generic technique to eliminate lookups

- Load all possible values from the table
- Use arithmetic (similar as for elimination of conditional branches) to pick the right one
- This is very slow for many table entries
Bitslicing

- Every algorithm can be implemented with just AND and XOR gates
- Idea: Simulate this in software:
  - Split each \( n \)-bit value across \( n \) registers (one bit per register)
  - Operate on registers with AND and XOR instructions (may also use OR, NEG, NAND, etc. if available)
  - This is very slow, because we use only one bit of a register but arithmetic is performed on all register bits in parallel
  - Perform \( m \) computations in parallel, where \( m \) is the register width
  - In other words: Treat \( m \)-bit registers as vector registers containing \( m \) elements of a single bit
  - This can be very fast if there are \( m \) independent data streams that all want the same computations
  - Performance highly depends on the algorithm and the microarchitecture
  - Some overhead for transforming input data to bitsliced representation (transpose data)
Bitslicing

- Every algorithm can be implemented with just AND and XOR gates
- Idea: Simulate this in software:
  - Split each $n$-bit value across $n$ registers (one bit per register)
  - Operate on registers with AND and XOR instructions (may also use OR, NEG, NAND, etc. if available)

- Performance highly depends on the algorithm and the microarchitecture
- Some overhead for transforming input data to bitsliced representation (transpose data)
Bitslicing

- Every algorithm can be implemented with just AND and XOR gates
- Idea: Simulate this in software:
  - Split each $n$-bit value across $n$ registers (one bit per register)
  - Operate on registers with AND and XOR instructions (may also use OR, NEG, NAND, etc. if available)
  - This is very slow, because we use only one bit of a register but arithmetic is performed on all register bits in parallel
Bitslicing

- Every algorithm can be implemented with just AND and XOR gates
- Idea: Simulate this in software:
  - Split each $n$-bit value across $n$ registers (one bit per register)
  - Operate on registers with AND and XOR instructions (may also use OR, NEG, NAND, etc. if available)
  - This is very slow, because we use only one bit of a register but arithmetic is performed on all register bits in parallel
  - Perform $m$ computations in parallel, where $m$ is the register width
  - In other words: Treat $m$-bit registers as vector registers containing $m$ elements of a single bit
Bitslicing

- Every algorithm can be implemented with just AND and XOR gates
- Idea: Simulate this in software:
  - Split each $n$-bit value across $n$ registers (one bit per register)
  - Operate on registers with AND and XOR instructions (may also use OR, NEG, NAND, etc. if available)
  - This is very slow, because we use only one bit of a register but arithmetic is performed on all register bits in parallel
  - Perform $m$ computations in parallel, where $m$ is the register width
  - In other words: Treat $m$-bit registers as vector registers containing $m$ elements of a single bit
- This can be very fast if there are $m$ independent data streams that all want the same computations
Bitslicing

- Every algorithm can be implemented with just AND and XOR gates
- Idea: Simulate this in software:
  - Split each \( n \)-bit value across \( n \) registers (one bit per register)
  - Operate on registers with AND and XOR instructions (may also use OR, NEG, NAND, etc. if available)
  - This is very slow, because we use only one bit of a register but arithmetic is performed on all register bits in parallel
  - Perform \( m \) computations in parallel, where \( m \) is the register width
  - In other words: Treat \( m \)-bit registers as vector registers containing \( m \) elements of a single bit
- This can be very fast if there are \( m \) independent data streams that all want the same computations
- Performance highly depends on the algorithm and the microarchitecture
Every algorithm can be implemented with just AND and XOR gates

Idea: Simulate this in software:
- Split each \( n \)-bit value across \( n \) registers (one bit per register)
- Operate on registers with AND and XOR instructions (may also use OR, NEG, NAND, etc. if available)
- This is very slow, because we use only one bit of a register but arithmetic is performed on all register bits in parallel
- Perform \( m \) computations in parallel, where \( m \) is the register width
- In other words: Treat \( m \)-bit registers as vector registers containing \( m \) elements of a single bit
- This can be very fast if there are \( m \) independent data streams that all want the same computations
- Performance highly depends on the algorithm and the microarchitecture
- Some overhead for transforming input data to bitsliced representation (transpose data)
Bitslicing AES

- Consider a “good” architecture for bitslicing
- For example, Intel Core 2: 3 bit-logical operations on 128-bit registers each cycle (384 bit operations per cycle!)
Bitslicing AES

- Consider a “good” architecture for bitslicing
- For example, Intel Core 2: 3 bit-logical operations on 128-bit registers each cycle (384 bit operations per cycle!)
- Obvious approach: Consider 128 independent input blocks
 Bitslicing AES

- Consider a “good” architecture for bitslicing
- For example, Intel Core 2: 3 bit-logical operations on 128-bit registers each cycle (384 bit operations per cycle!)
- Obvious approach: Consider 128 independent input blocks
- But: Core part of AES (SubBytes) is already $16 \times$ parallel
- Better for small packets: Consider just 8 independent blocks
- Need to pay some attention in ShiftRows and MixColumns
Bitslicing AES

- Consider a “good” architecture for bitslicing
- For example, Intel Core 2: 3 bit-logical operations on 128-bit registers each cycle (384 bit operations per cycle!)
- Obvious approach: Consider 128 independent input blocks
- But: Core part of AES (SubBytes) is already $16 \times$ parallel
- Better for small packets: Consider just 8 independent blocks
- Need to pay some attention in ShiftRows and MixColumns
- Start with a good hardware implementation of SubBytes (inversion in $\mathbb{F}_{2^8}$): Canright, 2005; Boyar, Peralta, 2009: 117 gates
Bitslicing AES

- Consider a “good” architecture for bitslicing
- For example, Intel Core 2: 3 bit-logical operations on 128-bit registers each cycle (384 bit operations per cycle!)
- Obvious approach: Consider 128 independent input blocks
- But: Core part of AES (SubBytes) is already $16 \times$ parallel
- Better for small packets: Consider just 8 independent blocks
- Need to pay some attention in ShiftRows and MixColumns
- Start with a good hardware implementation of SubBytes (inversion in $\mathbb{F}_{2^8}$): Canright, 2005; Boyar, Peralta, 2009: 117 gates
- Small problems: AMD64 instruction set only has 2-operand instructions, only 16 128-bit registers
- Software implementation needs some more operations than hardware implementation needs gates
Bitslicing AES

- Consider a “good” architecture for bitslicing
- For example, Intel Core 2: 3 bit-logical operations on 128-bit registers each cycle (384 bit operations per cycle!)
- Obvious approach: Consider 128 independent input blocks
- But: Core part of AES (SubBytes) is already $16 \times$ parallel
- Better for small packets: Consider just 8 independent blocks
- Need to pay some attention in ShiftRows and MixColumns
- Start with a good hardware implementation of SubBytes (inversion in $\mathbb{F}_{2^8}$): Canright, 2005; Boyar, Peralta, 2009: 117 gates
- Small problems: AMD64 instruction set only has 2-operand instructions, only 16 128-bit registers
- Software implementation needs some more operations than hardware implementation needs gates
- Käsper, Schwabe in 2009: 7.58 cycles/byte on Intel Core 2 Q9550 (bitsliced)
- Previously fastest: Bernstein, Schwabe in 2008: 10.58 cycles/byte (with table lookups)
Levels of optimization

- Consider the example of elliptic-curve cryptography
- Various levels of optimization:
  - Choice of scalar-multiplication algorithm
  - Choice of curve an underlying finite field
  - Choice of coordinates and addition and doubling formulas
  - Representation of finite-field elements in machine words and related algorithms (e.g. schoolbook vs. Karatsuba multiplication)
  - Low-level optimizations of machine instructions
- These levels are not independent, many subtle interactions
Finite-field arithmetic

- With enough parallel data streams, bitslicing seems to be good for binary-field arithmetic
Finite-field arithmetic

- With enough parallel data streams, bitslicing seems to be good for binary-field arithmetic
- In the context of elliptic-curve cryptography: Bernstein, 2009
Finite-field arithmetic

- With enough parallel data streams, bitslicing seems to be good for binary-field arithmetic
- In the context of elliptic-curve cryptography: Bernstein, 2009
- On most processors there is no instruction for multiplication of binary polynomials
Finite-field arithmetic

- With enough parallel data streams, bitslicing seems to be good for binary-field arithmetic
- In the context of elliptic-curve cryptography: Bernstein, 2009
- On most processors there is no instruction for multiplication of binary polynomials
- There are instructions to efficiently multiply 32-bit or 64-bit integers
Finite-field arithmetic

- With enough parallel data streams, bitslicing seems to be good for binary-field arithmetic
- In the context of elliptic-curve cryptography: Bernstein, 2009
- On most processors there is no instruction for multiplication of binary polynomials
- There are instructions to efficiently multiply 32-bit or 64-bit integers
- Obvious for large-prime fields: use these instructions
Finite-field arithmetic

- With enough parallel data streams, bitslicing seems to be good for binary-field arithmetic
- In the context of elliptic-curve cryptography: Bernstein, 2009
- On most processors there is no instruction for multiplication of binary polynomials
- There are instructions to efficiently multiply 32-bit or 64-bit integers
- Obvious for large-prime fields: use these instructions
- Consider the example of multiplication in $\mathbb{F}_p$ with $p = 2^{255} - 19$
Finite-field arithmetic

- With enough parallel data streams, bitslicing seems to be good for binary-field arithmetic
- In the context of elliptic-curve cryptography: Bernstein, 2009
- On most processors there is no instruction for multiplication of binary polynomials
- There are instructions to efficiently multiply 32-bit or 64-bit integers
- Obvious for large-prime fields: use these instructions
- Consider the example of multiplication in $\mathbb{F}_p$ with $p = 2^{255} - 19$
- Typical choice: break 255-bit integers into 4 chunks of 64 bits (or 8 chunks of 32 bits)
- For schoolbook multiplication: 16 $64 \times 64$-bit integer multiplications producing 128-bit results
Finite-field arithmetic

- With enough parallel data streams, bitslicing seems to be good for binary-field arithmetic
- In the context of elliptic-curve cryptography: Bernstein, 2009
- On most processors there is no instruction for multiplication of binary polynomials
- There are instructions to efficiently multiply 32-bit or 64-bit integers
- Obvious for large-prime fields: use these instructions
- Consider the example of multiplication in $\mathbb{F}_p$ with $p = 2^{255} - 19$
- Typical choice: break 255-bit integers into 4 chunks of 64 bits (or 8 chunks of 32 bits)
- For schoolbook multiplication: $16 \times 64 \times 64$-bit integer multiplications producing 128-bit results
- Obtain result (before reduction) in 8 64-bit chunks
Multiplication in $\mathbb{F}_{2^{255} - 19}$ on AMD64

```c
mulx0 = *(uint64 *)(xp + 0)
rax = *(uint64 *)(yp + 0)
(uint128) rdx rax = rax * mulx0
r0 = rax
r1 = rdx

rax = *(uint64 *)(yp + 8)
(uint128) rdx rax = rax * mulx0
carry? r1 += rax
r2 = 0
r2 += rdx + carry

rax = *(uint64 *)(yp + 16)
(uint128) rdx rax = rax * mulx0
carry? r2 += rax
r3 = 0
r3 += rdx + carry

rax = *(uint64 *)(yp + 24)
(uint128) rdx rax = rax * mulx0
carry? r3 += rax
r4 += rdx + carry
```

- Initialization: 4 multiplications, each with one addition and one add-with-carry
Multiplication in $\mathbb{F}_{2^{255}-19}$ on AMD64

\[
\begin{align*}
\text{mulx1} &= *(\text{uint64 } *)(\text{xp} + 8) \\
\text{rax} &= *(\text{uint64 } *)(\text{yp} + 0) \\
(\text{uint128}) \text{ rdx } \text{ rax} &= \text{ rax } \ast \text{ mulx1} \\
\text{carry?} \ r1 &= \text{ rax} \\
\text{mulc} &= 0 \\
\text{mulc} &= \text{ rdx } + \text{ carry} \\
\text{carry?} \ r1 &= \text{ rax} \\
\text{rdx} &= 0 + \text{ carry} \\
\text{carry?} \ r2 &= \text{ mulc} \\
\text{mulc} &= 0 \\
\text{mulc} &= \text{ rdx } + \text{ carry} \\
\text{carry?} \ r1 &= \text{ rax} \\
\text{rdx} &= 0 + \text{ carry} \\
\text{carry?} \ r2 &= \text{ mulc} \\
\text{mulc} &= 0 \\
\text{mulc} &= \text{ rdx } + \text{ carry} \\
\text{carry?} \ r1 &= \text{ rax} \\
\text{rdx} &= 0 + \text{ carry} \\
\text{carry?} \ r2 &= \text{ mulc} \\
\text{mulc} &= 0 \\
\text{mulc} &= \text{ rdx } + \text{ carry} \\
\end{align*}
\]

- Initialization: 4 multiplications, each with one addition and one add-with-carry
- Continue: Each multiplication comes with 2 adds-with-carry
Multiplication in $\mathbb{F}_{2^{255}-19}$ on AMD64

mulx1 = *(uint64 *)(xp + 8)
rax = *(uint64 *)(yp + 0)
(uint128) rdx rax = rax * mulx1
carry? r1 += rax
mulc = 0
mulc += rdx + carry

rax = *(uint64 *)(yp + 8)
(uint128) rdx rax = rax * mulx1
carry? r2 += rax
rdx += 0 + carry
carry? r2 += mulc
mulc = 0
mulc += rdx + carry

rax = *(uint64 *)(yp + 16)
(uint128) rdx rax = rax * mulx1
carry? r3 += rax
rdx += 0 + carry
carry? r3 += mulc
mulc = 0
mulc += rdx + carry

- Initialization: 4 multiplications, each with one addition and one add-with-carry
- Continue: Each multiplication comes with 2 adds-with-carry
- Intel Nehalem/Westmere: 3 additions per cycles, only 1 add-with-carry every two cycles
- Handling carries becomes a bottleneck!
A different representation

- Better approach: Chop 255-bit integers into 5 parts, radix $2^{51}$
- Schoolbook multiplication now needs 25 $64 \times 64$-bit multiplications
A different representation

- Better approach: Chop 255-bit integers into 5 parts, radix $2^{51}$
- Schoolbook multiplication now needs $25 \times 64 \times 64$-bit multiplications
- Intermediary results now have only 102 bits
A different representation

- Better approach: Chop 255-bit integers into 5 parts, radix $2^{51}$
- Schoolbook multiplication now needs $25 \times 64 \times 64$-bit multiplications
- Intermediary results now have only 102 bits
- Adding lower 64 bits still requires add-with-carry
- Adding upper parts only needs addition (carries go in extra space)
Better approach: Chop 255-bit integers into 5 parts, radix $2^{51}$
Schoolbook multiplication now needs $25 \times 64 \times 64$-bit multiplications
Intermediary results now have only 102 bits
Adding lower 64 bits still requires add-with-carry
Adding upper parts only needs addition (carries go in extra space)
We have verified that this is faster on Intel Nehalem/Westmere
General lesson: The obvious representation may not be the best
A different representation

- Better approach: Chop 255-bit integers into 5 parts, radix $2^{51}$
- Schoolbook multiplication now needs $25 \times 64 \times 64$-bit multiplications
- Intermediary results now have only 102 bits
- Adding lower 64 bits still requires add-with-carry
- Adding upper parts only needs addition (carries go in extra space)
- We have verified that this is faster on Intel Nehalem/Westmere

**General lesson:** The obvious representation may not be the best

- Bernstein, 2006: Use radix $2^{25.5}$ and double-precision floating-point multiplication on Pentium III
- Costigan, Schwabe 2009: Use radix $2^{12.75}$ and integer-vector multiplication on Cell SPU
- Bernstein, Schwabe 2012: Use radix $2^{25.5}$ and integer-vector multiplication on ARM NEON
A different representation

- Better approach: Chop 255-bit integers into 5 parts, radix $2^{51}$
- Schoolbook multiplication now needs 25 $64 \times 64$-bit multiplications
- Intermediary results now have only 102 bits
- Adding lower 64 bits still requires add-with-carry
- Adding upper parts only needs addition (carries go in extra space)
- We have verified that this is faster on Intel Nehalem/Westmere

**General lesson:** The obvious representation may not be the best

- Bernstein, 2006: Use radix $2^{25.5}$ and double-precision floating-point multiplication on Pentium III
- Costigan, Schwabe 2009: Use radix $2^{12.75}$ and integer-vector multiplication on Cell SPU
A different representation

- Better approach: Chop 255-bit integers into 5 parts, radix $2^{51}$
- Schoolbook multiplication now needs $25 \times 64 \times 64$-bit multiplications
- Intermediary results now have only 102 bits
- Adding lower 64 bits still requires add-with-carry
- Adding upper parts only needs addition (carries go in extra space)
- We have verified that this is faster on Intel Nehalem/Westmere

**General lesson:** The obvious representation may not be the best

- Bernstein, 2006: Use radix $2^{25.5}$ and double-precision floating-point multiplication on Pentium III
- Costigan, Schwabe 2009: Use radix $2^{12.75}$ and integer-vector multiplication on Cell SPU
- Bernstein, Schwabe 2012: Use radix $2^{25.5}$ and integer-vector multiplication on ARM NEON
Concluding remarks

qhasm

- Syntax I used for assembly is actually qhasm syntax
- Language developed by Bernstein to make programming on the assembly level easier: http://cr.yp.to/qhasm.html
Concluding remarks

qhasm

- Syntax I used for assembly is actually qhasm syntax
- Language developed by Bernstein to make programming on the assembly level easier: http://cr.yp.to/qhasm.html

Benchmarking your software

- Measuring performance of (cryptographic) software is easy to do wrong
- Please submit to eBACS: ECRYPT Benchmarking of Cryptographic Systems (Bernstein, Lange): http://bench.cr.yp.to
Concluding remarks

qhasm

▷ Syntax I used for assembly is actually qhasm syntax
▷ Language developed by Bernstein to make programming on the assembly level easier: http://cr.yp.to/qhasm.html

Benchmarking your software

▷ Measuring performance of (cryptographic) software is easy to do wrong
▷ Please submit to eBACS: ECRYPT Benchmarking of Cryptographic Systems (Bernstein, Lange): http://bench.cr.yp.to

Put software online

▷ A paper describing software is nice, it’s worth much more if it comes with the software
▷ Please make your software available!