The Certicom Challenges ECC2-X

Daniel V. Bailey, Brian Baldwin, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Gauthier van Damme, Giacomo de Meulenaer, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Christof Paar, Francesco Regazzoni, Peter Schwabe, and Leif Uhsadel

ECRYPT VAMPIRE I

September 9, 2009

SHARCS 2009

Overview

General set-up (Tanja Lange)

ASIC implementations (Frank Gurkaynak)

FPGA implementations (Daniel V. Bailey)

General-purpose CPU implementation (Daniel J. Bernstein)

Cell implementations (Peter Schwabe)

General set-up (Tanja Lange)

ASIC implementations (Frank Gurkaynak)

FPGA implementations (Daniel V. Bailey)

General-purpose CPU implementation (Daniel J. Bernstein)

Cell implementations (Peter Schwabe)

Certicom challenges

- The "exercises"
  - 79-bit: SOLVED December 1997
  - 89-bit: SOLVED February 1998
  - 97-bit: SOLVED September 1999
- Level I
  - ECC2K-108: SOLVED April 2000
  - ECCp-109: SOLVED Nov. 2002
  - ECC2-109: SOLVED April 2004
  - 131-bit: (ECC2K-130, ECC2-131, ECCp-131) still open
- Level II
  - 163-bit: (ECC2K-163, ECC2-163, ECCp-162) still open
  - 191-bit, 239-bit, 359-bit: still open

ECC2-XXX

- Our paper covers the binary challenges ECC2K-130, ECC2-131, ECC2K-163, and ECC2-163.
- The easiest of these is ECC2K-130, a Koblitz curve defined over $\mathbb{F}_{2^{131}}$.
- Challenge data for ECC2K-130:

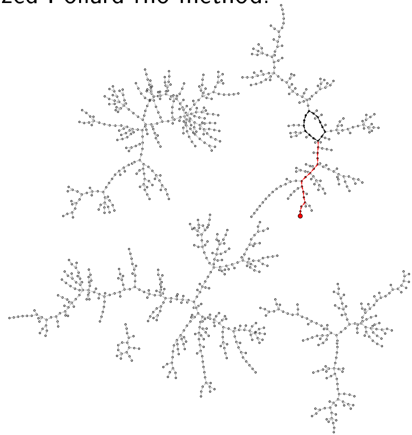
```
Px=05 1C99BFA6 F18DE467 C80C23B9 8C7994AA
Py=04 2EA2D112 ECEC71FC F7E000D7 EFC978BD
Qx=06 C997F3E7 F2C66A4A 5D2FDA13 756A37B1
Qy=04 A38D1182 9D32D347 BD0C0F58 4D546E9A
```

- Certicom:

  "The 109-bit Level I challenges are feasible using a very large network of computers. The 131-bit Level I challenges are expected to be infeasible against realistic software and hardware attacks, unless of course, a new algorithm for the ECDLP is discovered. The Level II challenges are infeasible given today's computer technology and knowledge."

DLPs on ECC

- No index-calculus-type attacks known for general elliptic curves.
- Pollard's rho method best generic attack (no memory needed).
- We have many platforms, each with many execution units.
  Use parallelized Pollard rho method:



- All units need to use the same step function and distinguished points.

Hardness of ECC2K-130

- Curve has cofactor 4.
- Koblitz curves are defined over $\mathbb{F}_2$ and thus the (small) Frobenius endomorphism operates on the $\mathbb{F}_{2^{131}}$-rational points. The operation is simply squaring the coordinates.
- Can define 'random' walk on classes under $\pm$ and Frobenius.
- Complexity of attack:

$$\sqrt{\frac{\pi \cdot 2^{131}}{2 \cdot 4 \cdot 2 \cdot 131}} \approx 2^{60.9}$$

  iterations ... provided that the the iteration works on the classes.
- Easy: P and $-P$ have same x coordinate.
- Harder: $x(P), x(P)^2, x(P)^{2^2}, \dots$ look quite different.
- Even more fun: can choose normal basis or polynomial basis representation of finite field; this changes the representation of the points.

Handling Frobenius

- In polynomial basis could compute all Frobenius powers and choose lexicographically smallest of these – but this needs 130 squarings and does not work well with normal basis.
- In normal basis, $x(P)$ and $x(P)^{2^j}$ have same Hamming weight. Convenient to use this. Polynomial basis has to convert for testing.
- Our step function:

  $$P_{i+1} = P_i \oplus \sigma^j(P_i),$$

  where $j = (\mathrm{HW}(x(P))/2 \bmod 8) + 3$.
- This nicely avoids short, fruitless cycles.
- Iteration consists of
  - converting $x(P)$ to normal basis (if necessary),
  - computing the Hamming weight $\mathrm{HW}(x(P))$ of the normal basis representation of $x(P)$,
  - checking that $\mathrm{HW}(x(P)) > 28$, computing j,
  - computing $P \oplus \sigma^j(P)$ (in the usual representation of P).
- Speed up by running multiple instances and combining inversion using Montgomery's trick.

General set-up (Tanja Lange)

ASIC implementations (Frank Gurkaynak)

FPGA implementations (Daniel V. Bailey)

General-purpose CPU implementation (Daniel J. Bernstein)

Cell implementations (Peter Schwabe)

Can we break ECC2K-130 using ASICs?

Our goals

- Determine the rough cost of the attack
- Find out if the attack is feasible using ASICs
- Provide an outline of what needs to be done.

Can we break ECC2K-130 using ASICs?

Our goals

- Determine the rough cost of the attack
- Find out if the attack is feasible using ASICs
- Provide an outline of what needs to be done.

What we did not do

- Exact implementation
- Address issues with off-chip communication for distinguished points

Our estimation methodology

- Select an affordable technology to implement the ASIC
- Take individual sub-components that will make up one step calculation
- Determine the post-layout performance limits of these sub-components.
- Leave healthy margins for real-life implementations.
- Compose one ASIC using multiple parallel instances that compute a single step
- Find out the performance obtained by a single ASIC
- Calculate how many ASICs you would need for such an attack

Cost and performance of 1 ASIC

- **Selected UMC 90nm**
  No special reason, could as well be any other technology.
- **MPW cost: 45,000 Euros**
  Standard cost for prototyping, no mass production. 200-300 dies can be produced this way.
- **Cost for packaging: 10,000 Euros**
  Cost for packaging roughly same for 50-250 dies.
- **Available core area: 2,000,000 gates**
  Total core area is 12mm^2. Space for I/O, PLL, Mem. etc
- **Internal clock speed: 1.2 - 1.5 GHz possible**
  I/O at 200-300 MHz, PLL required for internal clock.
- **Power is not issue in this project**
  Proper power distribution, heat removal required

Cost of calculating 1 step in the Pollard rho

For the ECC130-2K

- **Assuming normal basis**
  For a real application the tradeoff between the normal and polynomial basis should be investigated further.
- **One step consumes 1 inversion, 2 multiplications, 131 squarings and 1 multiple squarings**
  - This can be realized within 1,572 clock cycles
  - At the chosen technology, this function can be clocked as fast as 1.5 GHz
  - Can be implemented using 6,000 gates
- **More detailed numbers can be found in manuscript**
  Estimates were made with post-layout numbers

Cost of Attack

- **One ASIC can support 300-400 cores**
  Leaving room for PLLs, I/O, room for distinguished point evaluation.
- **Clock rate 1.25 GHz**
  Conservative estimation.
- One ASIC will have a throughput of 300-400 Million steps per second
- **I/O bandwidth of one chip will be around 30 Gb/s**
  Should be sufficient for point distribution
- To attack ECC2K-130 in one year approx. 69,000 Million steps per second are required
  This throughput can be achieved by 200-300 ASICs

Conclusions

- **ASIC implementation possible with reasonable cost**
  Around 200 ASICs, costing less than 60.000 Euros will be able to mount a successful attack in a year
- **Currently no one is working on a concrete implementation**
  These numbers suggest that the project is feasible, however, at the moment we do not have someone working on the project.
- **Practical implementation will be even faster**
  As soon as, someone starts working in earnest, more efficient implementations will almost certainly be developed.
- **Practical implementation will also suffer from technical issues**
  Such as I/O and memory bandwidth, overall routing etc. The last two points will probably balance each other out

General set-up (Tanja Lange)

ASIC implementations (Frank Gurkaynak)

FPGA implementations (Daniel V. Bailey)

General-purpose CPU implementation (Daniel J. Bernstein)

Cell implementations (Peter Schwabe)

COPACOBANA

- A battery of low-cost FPGAs aimed at high-computation, low-communication tasks
- Cost-optimized parallel code breaker introduced (Copa) at SHARCS 2006
- New and Improved for 2009: COPA5000
- Contains 128 Spartan-3 5000 FPGAs (XC3S5000-4FG676)
- Faster communication infrastructure and 32MB of external RAM per FPGA

How Best to Use Copa?

- 1 inversion, 2 mults, 1 squaring, 1 repeated-squaring needed for one step of the Rho method
- As with the Cell implementation, two teams
- One implementation operates on elements in polynomial basis and converts to check if a DP has been generated
- Another operates directly in normal basis – no need to convert
- Which is a better fit for Copa (time-area product)?

Polynomial Basis

- More literature on PB: generally beats NB for efficient implementation
- But attacking ECC2K-130 is different: the Frobenius map is free in NB
- PB implementation aims for the best of both worlds: faster PB multiplication followed by conversion and Frobenius
- Engine uses Montgomery's trick to process 64 inversions simultaneously
- Engine Total: 3,656 slices, 1,468 slices for multiplier, 75 slices for square, 1,206 slices for conversion
- 9 engines can fit in one FPGA, yielding 23.4 DPs/day

Normal Basis

- Normal Basis has fast squaring and Frobenius
- But multiplication is much more expensive
- Inversion uses Itoh-Tsujii, (8 multiplications!) so the design task becomes keeping the inversion unit busy
- 32 inversions simultaneously: then 32 dedicated multipliers recover individual inverses
- Only 4 engines fit on-chip, but one chip still yields an estimated 24 DPs/day
- Next step: better multiplication!

General set-up (Tanja Lange)

ASIC implementations (Frank Gurkaynak)

FPGA implementations (Daniel V. Bailey)

General-purpose CPU implementation (Daniel J. Bernstein)

Cell implementations (Peter Schwabe)

What about software?

Have an implementation for the amd64 architecture.

What about software?

Have an implementation for the amd64 architecture.

What about software?

Have an implementation for the amd64 architecture.
Architecture provides 16 128-bit vector registers.
Two-operand vector instructions: `a ^= b`, `a &= b`, etc.

Some targeted CPUs:

- 2200MHz 4-core AMD Phenom 9550 100f23.
- 2210MHz 2-core AMD Opteron 875 20f10.
- 2404MHz 4-core Intel Core 2 Q6600 6fb.
- 2668MHz 4-core Intel Core i7 920 106a4.
- 3000MHz 4-core Intel Core 2 Q6850 6fb.

Initial focus: Core 2. Each core has 3 ALUs.
Each ALU does ≤ 1 vector operation per cycle.

Bitslicing

```
f0 = 1;
f1 = 0;
g0 = 1;
g1 = 1;

c = f0 & g1;
d = f1 & g0;
h0 = f0 & g0;
h1 = c ^ d;
h2 = f1 & g1;
```

5 bit operations.

Bitslicing

```
f0 = 1;
f1 = 0;
g0 = 1;
g1 = 1;

c = f0 & g1;
d = f1 & g0;
h0 = f0 & g0;
h1 = c ^ d;
h2 = f1 & g1;
```

5 bit operations.

```
f0 = 1;
f1 = 1;
g0 = 0;
g1 = 1;

c = f0 & g1;
d = f1 & g0;
h0 = f0 & g0;
h1 = c ^ d;
h2 = f1 & g1;
```

5 bit operations.

Bitslicing

```
f0 = 1;
f1 = 0;
g0 = 1;
g1 = 1;

c = f0 & g1;
d = f1 & g0;
h0 = f0 & g0;
h1 = c ^ d;
h2 = f1 & g1;
```

5 bit operations.

```
f0 = 1;
f1 = 1;
g0 = 0;
g1 = 1;

c = f0 & g1;
d = f1 & g0;
h0 = f0 & g0;
h1 = c ^ d;
h2 = f1 & g1;
```

5 bit operations.

```
f0 = 0;
f1 = 1;
g0 = 0;
g1 = 1;

c = f0 & g1;
d = f1 & g0;
h0 = f0 & g0;
h1 = c ^ d;
h2 = f1 & g1;
```

5 bit operations.

Bitslicing

```
f0 = bitvector(1,1,0);
f1 = bitvector(0,1,1);
g0 = bitvector(1,0,0);
g1 = bitvector(1,1,1);

c = f0 & g1;
d = f1 & g0;
h0 = f0 & g0;
h1 = c ^ d;
h2 = f1 & g1;
```

5 vector operations.

Counting bit operations for ECC2K-130

Software represents field element as 131 bits in poly basis:
$f_0, f_1, \ldots, f_{130}$ represents $\sum_i f_i x^i \bmod x^{131} + x^{13} + x^2 + x + 1$.

Costs of arithmetic as implemented —

- 14149 bit ops for $f, g \mapsto fg$.

Counting bit operations for ECC2K-130

Software represents field element as 131 bits in poly basis:
$f_0, f_1, \ldots, f_{130}$ represents $\sum_i f_i x^i \bmod x^{131} + x^{13} + x^2 + x + 1$.

Costs of arithmetic as implemented —

- 14149 bit ops for $f, g \mapsto fg$.
- 203 bit ops for $f \mapsto f^2$.

Counting bit operations for ECC2K-130

Software represents field element as 131 bits in poly basis:
$f_0, f_1, \ldots, f_{130}$ represents $\sum_i f_i x^i$ mod $x^{131} + x^{13} + x^2 + x + 1$.

Costs of arithmetic as implemented —

- 14149 bit ops for $f, g \mapsto fg$.
- 203 bit ops for $f \mapsto f^2$.
- 3380 bit ops for conversion to normal basis.
  `http://binary.cr.yp.to/linearmod2.html`

Counting bit operations for ECC2K-130

Software represents field element as 131 bits in poly basis:
$f_0, f_1, \ldots, f_{130}$ represents $\sum_i f_i x^i$ mod $x^{131} + x^{13} + x^2 + x + 1$.

Costs of arithmetic as implemented —

- 14149 bit ops for $f, g \mapsto fg$.
- 203 bit ops for $f \mapsto f^2$.
- 3380 bit ops for conversion to normal basis.
  `http://binary.cr.yp.to/linearmod2.html`
- 393 bit ops for $f, g, ? \mapsto f + ?(g - f)$.

Counting bit operations for ECC2K-130

Software represents field element as 131 bits in poly basis:
$f_0, f_1, \ldots, f_{130}$ represents $\sum_i f_i x^i \bmod x^{131} + x^{13} + x^2 + x + 1$.

Costs of arithmetic as implemented —

- 14149 bit ops for $f, g \mapsto fg$.
- 203 bit ops for $f \mapsto f^2$.
- 3380 bit ops for conversion to normal basis.
  `http://binary.cr.yp.to/linearmod2.html`
- 393 bit ops for $f, g, ? \mapsto f + ?(g - f)$.
- 139582 bit ops for $f \mapsto 1/f$.

Counting bit operations for ECC2K-130

Software represents field element as 131 bits in poly basis:
$f_0, f_1, \ldots, f_{130}$ represents $\sum_i f_i x^i$ mod $x^{131} + x^{13} + x^2 + x + 1$.

Costs of arithmetic as implemented —

- 14149 bit ops for $f, g \mapsto fg$.
- 203 bit ops for $f \mapsto f^2$.
- 3380 bit ops for conversion to normal basis.
  `http://binary.cr.yp.to/linearmod2.html`
- 393 bit ops for $f, g, ? \mapsto f + ?(g - f)$.
- 139582 bit ops for $f \mapsto 1/f$.
- 131 bit ops for $f, g \mapsto f + g$.

Counting bit operations for ECC2K-130

Software represents field element as 131 bits in poly basis:
$f_0, f_1, \ldots, f_{130}$ represents $\sum_i f_i x^i \bmod x^{131} + x^{13} + x^2 + x + 1$.

Costs of arithmetic as implemented —

- 14149 bit ops for $f, g \mapsto fg$.
- 203 bit ops for $f \mapsto f^2$.
- 3380 bit ops for conversion to normal basis.
  `http://binary.cr.yp.to/linearmod2.html`
- 393 bit ops for $f, g, ? \mapsto f + ?(g - f)$.
- 139582 bit ops for $f \mapsto 1/f$.
- 131 bit ops for $f, g \mapsto f + g$.
- 654 bit ops for weight computation, comparison.

Counting bit operations for ECC2K-130

Software represents field element as 131 bits in poly basis:
$f_0, f_1, \ldots, f_{130}$ represents $\sum_i f_i x^i \bmod x^{131} + x^{13} + x^2 + x + 1$.

Costs of arithmetic as implemented — batching 48 inversions:

- 14149 bit ops for $f, g \mapsto fg$. $\times 5 = 70745$
- 203 bit ops for $f \mapsto f^2$. $\times 21 = 4263$
- 3380 bit ops for conversion to normal basis. $\times 1 = 3380$
  `http://binary.cr.yp.to/linearmod2.html`
- 393 bit ops for $f, g, ? \mapsto f + ?(g - f)$. $\times 6 = 2358$
- 139582 bit ops for $f \mapsto 1/f$. $(\cdots - 3\mathrm{M})/48 = 2024$
- 131 bit ops for $f, g \mapsto f + g$. $\times 7 = 917$
- 654 bit ops for weight computation, comparison. $\times 1 = 654$

Counting cycles for ECC2K-130

84341 bit ops for iteration. Confirmed by computer.
84341 vector ops handle 128 parallel iterations.
On one core: $\geq 84341/3$ cycles for 128 iterations; i.e.,
≥ 219 cycles per iteration.

Counting cycles for ECC2K-130

84341 bit ops for iteration. Confirmed by computer.
84341 vector ops handle 128 parallel iterations.
On one core: $\geq 84341/3$ cycles for 128 iterations; i.e.,
≥ 219 cycles per iteration.

3GHz Core 2 Q6850 actually uses 694 cycles per iteration.
4 cores: 17.29 M iterations/sec. 3943 CPUs: done in 1 year.

Main bottleneck: loads, stores. **Need better scheduling!**
Other directions for improvements:

- Faster poly mult. Should save $\approx 10\%$.
- Faster reduction. Try $x^{131} + x^{36} + x^{27} + x^{18} + 1$.
- Normal-basis mult. Use 2007 vzG–Shokrollahi[2].
- Larger batch size. Make sure to prefetch from DRAM.

General set-up (Tanja Lange)

ASIC implementations (Frank Gurkaynak)

FPGA implementations (Daniel V. Bailey)

General-purpose CPU implementation (Daniel J. Bernstein)

Cell implementations (Peter Schwabe)

The Cell Broadband Engine

Well known architecture (from the previous talk)

The Cell Broadband Engine

Well known architecture (from the previous talk)

The Cell's SPUs

- Running at 3.2 GHz
- Register file with 128 128-bit registers
- All arithmetic instructions are SIMD instructions
- At most one arithmetic instruction per cycle
- At most one load/store instruction per cycle
- The Playstation makes 6 of these SPUs available

The Cell Broadband Engine

Well known architecture (from the previous talk)

The Cell's SPUs

- Running at 3.2 GHz
- Register file with 128 128-bit registers
- All arithmetic instructions are SIMD instructions
- At most one arithmetic instruction per cycle
- At most one load/store instruction per cycle
- The Playstation makes 6 of these SPUs available

"Fast 128-bit vector operations $\Longrightarrow$ bitsliced implementation?"

Shall we go bitsliced?

- Bitsliced implementation requires more memory (because we always have to store 128 values)
- Only *one* arithmetic instruction per cycle
- Cell's SPUs do in-order execution
- Unrolling and inlining yield huge speed-ups (but increase code size)
- "Everything" (code, data segment, stack, heap) has to fit into 256 KB of local storage.

Shall we go bitsliced?

- Bitsliced implementation requires more memory (because we always have to store 128 values)
- Only *one* arithmetic instruction per cycle
- Cell's SPUs do in-order execution
- Unrolling and inlining yield huge speed-ups (but increase code size)
- "Everything" (code, data segment, stack, heap) has to fit into 256 KB of local storage.

$\Longrightarrow$ It's not obvious that bitsliced implementations are faster

$\Longrightarrow$ Two teams independently implemented bitsliced and non-bitsliced

Cycles per “step” on one SPU

not bitsliced

- 31 Jul: 2565

bitsliced

Cycles per “step” on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735

bitsliced

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735

bitsliced

- 06 Aug: 6488

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587
- 13 Aug: 1389

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735
- 19 Aug: 1426

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587
- 13 Aug: 1389

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735
- 19 Aug: 1426
- 19 Aug: 1293

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587
- 13 Aug: 1389

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735
- 19 Aug: 1426
- 19 Aug: 1293

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587
- 13 Aug: 1389
- 30 Aug: 1180

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735
- 19 Aug: 1426
- 19 Aug: 1293
- 04 Sep: 1157

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587
- 13 Aug: 1389
- 30 Aug: 1180

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735
- 19 Aug: 1426
- 19 Aug: 1293
- 04 Sep: 1157

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587
- 13 Aug: 1389
- 30 Aug: 1180
- 5 Sep: 1051

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735
- 19 Aug: 1426
- 19 Aug: 1293
- 04 Sep: 1157

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587
- 13 Aug: 1389
- 30 Aug: 1180
- 5 Sep: 1051
- 7 Sep: 1047

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735
- 19 Aug: 1426
- 19 Aug: 1293
- 04 Sep: 1157
- Next week?

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587
- 13 Aug: 1389
- 30 Aug: 1180
- 5 Sep: 1051
- 7 Sep: 1047
- Next week?

Cycles per "step" on one SPU

not bitsliced

- 31 Jul: 2565
- 03 Aug: 1735
- 19 Aug: 1426
- 19 Aug: 1293
- 04 Sep: 1157
- Next week?

bitsliced

- 06 Aug: 6488
- 10 Aug: 1587
- 13 Aug: 1389
- 30 Aug: 1180
- 5 Sep: 1051
- 7 Sep: 1047
- Next week?

$\Longrightarrow$ Currently: < 3800 PS3 years for ECC2K-130

Thank you for your attention.