Breaking ECC2K-130
(on Cell CPUs and NVIDIA GPUs)

Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner,
Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng,
Gauthier van Damme, Giacomo de Meulenaer,
Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu,
Frank Gürkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens,
Ruben Niederhagen, Christof Paar, Francesco Regazzoni,
Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, Bo-Yin Yang

April 2, 2010

CARAMEL seminar, INRIA Nancy

How hard is the ECDLP?

The ECDLP

Given an elliptic curve E over a finite field $\mathbb{F}_q$ and two points $P \in E(\mathbb{F}_q)$ and $Q \in \langle P \rangle$, find k, such that $Q = [k]P$.

How hard is the ECDLP?

The ECDLP

Given an elliptic curve E over a finite field $\mathbb{F}_q$ and two points $P \in E(\mathbb{F}_q)$ and $Q \in \langle P \rangle$, find k, such that $Q = [k]P$.

- Standard answer (for most elliptic curves): Solving ECDLP takes $O(\sqrt{n})$, where $n = |\langle P \rangle|$, if n is prime
- Best known algorithm if n is prime: Pollard's rho algorithm, running time: $O(\sqrt{n})$
- Problem: O-notation hides all linear factors

How hard is the ECDLP?

The ECDLP

Given an elliptic curve E over a finite field $\mathbb{F}_q$ and two points $P \in E(\mathbb{F}_q)$ and $Q \in \langle P \rangle$, find k, such that $Q = [k]P$.

- Standard answer (for most elliptic curves): Solving ECDLP takes $O(\sqrt{n})$, where $n = |\langle P \rangle|$, if n is prime
- Best known algorithm if n is prime: Pollard's rho algorithm, running time: $O(\sqrt{n})$
- Problem: O-notation hides all linear factors

Rephrase the question

Given an elliptic curve E and two points P and Q as above and given a number of computers (or FPGAs, or ASICS, or money), how much time does it take to solve the specific ECDLP?

The Certicom challenges

1997: Certicom announces several ECDLP prizes:

The Challenge is to compute the ECC private keys from the given list of ECC public keys and associated system parameters. This is the type of problem facing an adversary who wishes to completely defeat an elliptic curve cryptosystem.

Objectives:

1. To increase the cryptographic community's understanding and appreciation of the difficulty of the ECDLP.

2. To confirm comparisons of the security levels of systems such as ECC, RSA and DSA that have been made based primarily on theoretical considerations.

The Certicom challenges (ctd.)

3. To provide information on how users of elliptic curve public-key cryptosystems should select suitable key lengths for a desired level of security.

4. To determine whether there is any significant difference in the difficulty of the ECDLP for elliptic curves over $\mathbb{F}_{2^m}$ and the ECDLP for elliptic curves over $\mathbb{F}_p$.

5. To determine whether there is any significant difference in the difficulty of the ECDLP for random elliptic curves over $\mathbb{F}_{2^m}$ and the ECDLP for Koblitz curves.

6. To encourage and stimulate research in computational and algorithmic number theory and, in particular, the study of the ECDLP.

Three levels of challenges

Level-0 challenges – exercises

Challenges of 79 bits, 89 bits, and 97 bits (size of $E(\mathbb{F}_q)$).

Level-0 challenges have all been solved

Level-1 challenges

Challenges of 109 bits, and 131 bits.

109-bit challenges have all been solved, 131-bit challenges have all *not* been solved, yet.

Level-2 challenges

Challenges of 163 bits, 191 bits, 239 bits, and 359 bits.

Level-2 challenges have all not been solved, yet.

The "next" open challenge: ECC2K-130

ECC2K-130

Elliptic curve E is the Koblitz curve $y^2 + xy = x^3 + 1$ over $\mathbb{F}_{2^{131}} = \mathbb{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$
The order of P is $680564733841876926932320129493409985129 \approx 2^{129}$.

Claimed hardness of ECC2K-130

> *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.*

(from Certicom's description of the challenges)

Parallelized Pollard's rho algorithm

- Algorithm by van Oorschot and Wiener
- Declare an easy-to-recognize subset of $|\langle P \rangle|$ as *distinguished*
- Use client-server infrastructure

Parallelized Pollard's rho algorithm

- Algorithm by van Oorschot and Wiener
- Declare an easy-to-recognize subset of $|\langle P \rangle|$ as *distinguished*
- Use client-server infrastructure
- Client:
  - Generate random point $R_0 = [a_0]P + [b_0]Q$ from random seed s
  - Apply pseudo-random iteration function f to obtain $R_{i+1} = f(R_i)$
  - When a distinguished point R_d is reached: Send (s, R_d) to the server
  - Generate new random input point

Parallelized Pollard's rho algorithm

- Algorithm by van Oorschot and Wiener
- Declare an easy-to-recognize subset of $|\langle P \rangle|$ as *distinguished*
- Use client-server infrastructure
- Client:
  - Generate random point $R_0 = [a_0]P + [b_0]Q$ from random seed s
  - Apply pseudo-random iteration function f to obtain $R_{i+1} = f(R_i)$
  - When a distinguished point R_d is reached: Send (s, R_d) to the server
  - Generate new random input point
- Server:
  - Search incoming distinguished points for duplicates (*collision*)
  - Use the information about the starting points (random seed) to obtain $R_d = [a_d]P + [b_d]Q$ and $R_d = [c_d]P + [d_d]Q$
  - Compute solution

$$Q = \frac{c_d - a_d}{d_d - b_d} P$$

.

Parallelized Pollard's rho algorithm

- Algorithm by van Oorschot and Wiener
- Declare an easy-to-recognize subset of $|\langle P \rangle|$ as *distinguished*
- Use client-server infrastructure
- Client:
  - Generate random point $R_0 = [a_0]P + [b_0]Q$ from random seed s
  - Apply pseudo-random iteration function f to obtain $R_{i+1} = f(R_i)$
  - When a distinguished point R_d is reached: Send (s, R_d) to the server
  - Generate new random input point
- Server:
  - Search incoming distinguished points for duplicates (*collision*)
  - Use the information about the starting points (random seed) to obtain $R_d = [a_d]P + [b_d]Q$ and $R_d = [c_d]P + [d_d]Q$
  - Compute solution
    $$Q = \frac{c_d - a_d}{d_d - b_d} P$$
    .
- Requires iteration function to preserve knowledge about the linear combination in P and Q.

How do we choose f?

"Adding walks"

Define $f(R_i) = R_i + [c_r]P + [d_r]Q$ where $r = h(R_i)$.

Iteration modulo negation

- P and $-P$ have same x-coordinate. Search for x-coordinate collision.
- Halves the size of the search space, thus factor-$\sqrt{2}$ speedup
- Requires that $f(P_i) = f(-P_i)$.
- For example use: $f(R_i) = |R_i| + [c_r]P + [d_r]Q$ with $|R_i|$ as, e.g., lexicographic minimum of $R_i, -R_i$.
- Comes with some problems (fruitless cycles), not exactly factor-$\sqrt{2}$ speedup

How do we choose f (Part II)

- Negation is an efficiently computable endomorphism
- Koblitz curves have other efficiently computable endomorphisms: powers of the Frobenius $\sigma^j(x, y) = (x^{2^j}, y^{2^j})$
- In our case 130 such endomorphisms
- Idea: Let f operate on equivalence classes modulo $\pm\sigma^j$
- In total: Save a factor of $\sqrt{2 \cdot 131}$

Our choice

Distinguished points

We call a point $R = (x_R, y_R)$ distinguished, if $\mathsf{HW}(x_R)$ (the Hamming weight of x_R in normal-basis representation) is ≤ 34.

Iteration function

Our iteration function is

$$R_{i+1} = f(R_i) = \sigma^j(R_i) + R_i,$$

where σ is the Frobenius endomorphism and

$$j = ((\mathsf{HW}(x_{R_i})/2) \pmod 8) + 3.$$

Computing the iteration function

$$R_{i+1} = f(R_i) = \sigma^j(R_i) + R_i,$$

- One elliptic curve addition
- One application of σ^j
- One conversion to normal-basis representation
- One Hamming-weight computation

Computing the iteration function

$$R_{i+1} = f(R_i) = \sigma^j(R_i) + R_i,$$

- One elliptic curve addition
  - we use affine coordinates
  - 2 multiplications, 1 squaring, 6 additions and 1 inversion
- One application of σ^j
- One conversion to normal-basis representation
- One Hamming-weight computation

Computing the iteration function

$$R_{i+1} = f(R_i) = \sigma^j(R_i) + R_i,$$

- One elliptic curve addition
  - we use affine coordinates
  - 2 multiplications, 1 squaring, 6 additions and 1 inversion
- One application of σ^j
  - Two computations of the form x^{2^m} for $3 \leq m \leq 10$ (m-squaring)
- One conversion to normal-basis representation
- One Hamming-weight computation

Computing the iteration function

$$R_{i+1} = f(R_i) = \sigma^j(R_i) + R_i,$$

- One elliptic curve addition
  - we use affine coordinates
  - 2 multiplications, 1 squaring, 6 additions and 1 inversion
- One application of σ^j
  - Two computations of the form x^{2^m} for $3 \leq m \leq 10$ (m-squaring)
- One conversion to normal-basis representation
- One Hamming-weight computation
- Inversions can be batched and performed using Montgomery's trick
- For large batch: Trade one inversion for 3 multiplications

Implementing the iteration function
on the Cell Broadband Engine (Playstation 3)

The technique of bitslicing

- Bernstein set new software speed records for batched binary-field arithmetic using bitslicing (CRYPTO 2009)
- Elements of $\mathbb{F}_{2^{131}}$ can be represented as a sequence of 131 bits
- Instead of putting these 131 bits in ,e.g., two 128-bit register, put them in 131 registers, one register per bit
- Perform arithmetic by simulating a hardware implementation using bit-logical instructions such as `AND` and `XOR`
- Inefficient for one field operation, but can process 128 batched operations in parallel (for 128-bit registers).
- Use spills to the stack to overcome lack of registers

Implementing the iteration function
on the Cell Broadband Engine (Playstation 3)

Is bitslicing really better?

- Bernstein's record was on the Intel Core 2, the Cell is different
- Cell SPU: Only 1 bit-logical operation per cycle (Core 2: 3 operations per cycle)
- Cell SPU: 128 128-bit registers (Core 2: 16 128-bit registers)
- Cell SPU can do one load or store per bit operation (Core 2: 1 load per 3 bit operations)
- Cell SPU has to fit all code and active data set in only 256 KB of *local storage*. Bitslicing requires more memory (because of the high level of parallelism)

Implementing the iteration function

on the Cell Broadband Engine (Playstation 3)

Is bitslicing really better?

- Bernstein's record was on the Intel Core 2, the Cell is different
- Cell SPU: Only 1 bit-logical operation per cycle (Core 2: 3 operations per cycle)
- Cell SPU: 128 128-bit registers (Core 2: 16 128-bit registers)
- Cell SPU can do one load or store per bit operation (Core 2: 1 load per 3 bit operations)
- Cell SPU has to fit all code and active data set in only 256 KB of *local storage*. Bitslicing requires more memory (because of the high level of parallelism)

Decision: Let's figure out what's best by implementing both, bitsliced and non-bitsliced, independently by two groups.

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 06 Aug: 6488 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 30 Aug: 1180 (bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- 05 Sep: 1051 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 30 Aug: 1180 (bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 30 Aug: 1180 (bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- We surrender!

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- We surrender!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- We surrender!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- We surrender!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)
- 14 Oct: 844 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- We surrender!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)
- 14 Oct: 844 (bitsliced)
- 15 Oct: 789 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- We surrender!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)
- 14 Oct: 844 (bitsliced)
- 15 Oct: 789 (bitsliced)
- 29 Oct: 749 (bitsliced)

What happened from 08/06 to 09/07?

From 6488 cycles to 1047 cycles

- Start with C++ implementation for the Core 2 (by Bernstein)
- Port to C (6488 cycles)
- Reimplement speed-critical parts in `qhasm`
- Most important: degree-130 polynomial multiplication

What happened from 08/06 to 09/07?

From 6488 cycles to 1047 cycles

- Start with C++ implementation for the Core 2 (by Bernstein)
- Port to C (6488 cycles)
- Reimplement speed-critical parts in `qhasm`
- Most important: degree-130 polynomial multiplication
  - Minimal number of bit operations: 11961 (`binary.cr.yp.to`)
  - Turn this into C code: doesn't compile
  - Decision: Sacrifice some bit operations
  - 2 levels of Karatsuba
  - Fast degree-32 polynomial multiplication (1286 bit operations)
  - Write scheduler to obtain code running in 1303 cycles (`qhasm`)
  - In total: 14503 cycles for degree-130 polynomial multiplication

What happened from 08/06 to 09/07?

From 6488 cycles to 1047 cycles

- Start with C++ implementation for the Core 2 (by Bernstein)
- Port to C (6488 cycles)
- Reimplement speed-critical parts in `qhasm`
- Most important: degree-130 polynomial multiplication
  - Minimal number of bit operations: 11961 (`binary.cr.yp.to`)
  - Turn this into C code: doesn't compile
  - Decision: Sacrifice some bit operations
  - 2 levels of Karatsuba
  - Fast degree-32 polynomial multiplication (1286 bit operations)
  - Write scheduler to obtain code running in 1303 cycles (`qhasm`)
  - In total: 14503 cycles for degree-130 polynomial multiplication
- Also implement Hamming-weight computation, squarings, conditional squarings, polynomial reduction in `qhasm`

What happened from 09/07 to 10/15?

From 1047 cycles to 789 cycles

- Start with polynomial-basis representation of elements
- How about normal-basis representation?
- Advantages:
  - m-squarings are just rotations
  - Conversion to normal-basis is free
- Disadvantage: Multiplications are slower

What happened from 09/07 to 10/15?

From 1047 cycles to 789 cycles

- Start with polynomial-basis representation of elements
- How about normal-basis representation?
- Advantages:
  - m-squarings are just rotations
  - Conversion to normal-basis is free
- Disadvantage: Multiplications are slower
- Shokrollahi et al.: Efficient conversion from type-2 normal basis to polynomial basis and back (WAIFI 2007), improvements by Bernstein and Lange
- Use this conversion, apply polynomial multiplcation, apply inverse conversion
- Conversion (of course) also implemented in `qhasm`
- Overhead for conversions is more than compensated by savings in m-squarings and basis conversion

What happened from 10/15 to 10/29?

From 789 cycles to 749 cycles

- Only 256 KB of local storage (LS): Batch size for Montgomery inversions of 14
- Idea: swap the active set of data between LS and main memory
- Has to be done explicitely using DMA transfers
- Transfers can be interleaved with computations $\Rightarrow$ almost no overhead
- Increase Montgomery batch size to 512

NVIDIA GPUs (work in progress)

- Started with a CUDA implementation
- Problem: CUDA compiler `nvcc` has problems with register allocation in large kernels
- Writing `ptx` doesn't help (register allocation is done *after* generating `ptx`)

NVIDIA GPUs (work in progress)

- Started with a CUDA implementation
- Problem: CUDA compiler `nvcc` has problems with register allocation in large kernels
- Writing `ptx` doesn't help (register allocation is done *after* generating `ptx`)
- Idea: Use `cudasm` and `decuda` (Reverse-engineered assembler and disassembler for G8x and G9x NVIDIA GPUs)
- Of course: Make `qhasm` support `cudasm` as target

NVIDIA GPUs (work in progress)

- Started with a CUDA implementation
- Problem: CUDA compiler `nvcc` has problems with register allocation in large kernels
- Writing `ptx` doesn't help (register allocation is done *after* generating `ptx`)
- Idea: Use `cudasm` and `decuda` (Reverse-engineered assembler and disassembler for G8x and G9x NVIDIA GPUs)
- Of course: Make `qhasm` support `cudasm` as target
- Impossible to optimize only speed-critical parts (no function calls possible!)
- Write the whole kernel (iteration function) in `qhasm`
- Early version had 125,824 lines of assembly code
- Now at 1379 cycles per iteration (with smaller code)

Results

Breaking ECC2K-130 in one year takes:

- 2462 Cell CPUs (Playstation 3), *or*
- 1262 NVIDIA GTX 295 graphic cards

Results

Breaking ECC2K-130 in one year takes:

- 2462 Cell CPUs (Playstation 3), *or*
- 1262 NVIDIA GTX 295 graphic cards,*or*
- 3039 3-GHz Core 2 CPUs, *or*
- 2026 XC3S5000 FPGAs.

Results

Breaking ECC2K-130 in one year takes:

- 2462 Cell CPUs (Playstation 3), *or*
- 1262 NVIDIA GTX 295 graphic cards,*or*
- 3039 3-GHz Core 2 CPUs, *or*
- 2026 XC3S5000 FPGAs.

That's what Certicom calls "infeasible"!

ECC2K-130 online

Progress of the attack: http://ecc-challenge.info
News: https://twitter.com/ECCchallenge

ECC2K-130 online

Progress of the attack: http://ecc-challenge.info
News: https://twitter.com/ECCchallenge

Papers

Breaking ECC2K-130:
http://eprint.iacr.org/2009/541/
ECC2K-130 on Cell CPUs:
http://eprint.iacr.org/2010/077/
Type-II Optimal Polynomial Bases:
http://eprint.iacr.org/2010/069/

... more on FPGAs and GPUs soon