Energy-Efficient ARM64 Cluster with Cryptanalytic Applications

80 Cores That Do Not Cost You an ARM and a Leg

Latincrypt 2017, 21st September 2017

 Thom Wiggers

iCIS | Digital Security
Radboud University

Outline

So you want to break crypto

1. Investigate attacks

So you want to break crypto

1. Investigate attacks
2. Implement attacks in software

So you want to break crypto

1. Investigate attacks
2. Implement attacks in software
3. ???

So you want to break crypto

1. Investigate attacks
2. Implement attacks in software
3. ???
4. Profit

So you want to break crypto

1. Investigate attacks
2. Implement attacks in software
3. Run software on hugely expensive clusters
4. Profit

Typical Platforms

“Desktop” CPUs

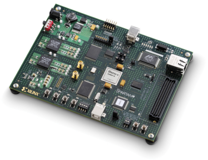
- Easy to program
- $$$$$
- Fairly high-power
- Fast with modern CPU extensions (SSE, AVX2)

GPUs

- Harder to program
- $$$$$
- Very high-power
- Much faster than CPUs on certain workloads

FPGAs

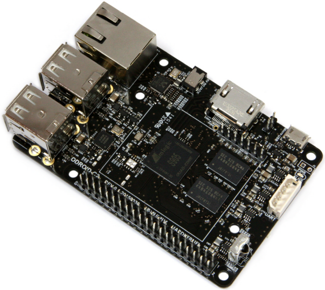
- Very hard to program
- $$$$$–$$$$$
- Low power
- Much, much faster than CPUs on certain workloads



Atypical platform

“Mobile” CPUs

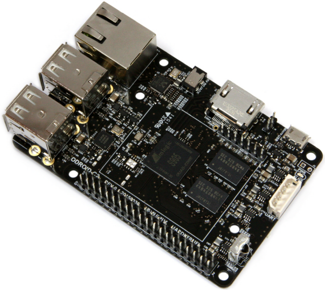
- Smartphones and IoT
- Easy to program for
- $$$$$
- Low power
- OK speeds?



ODROID-C2 devboard

ODROID-C2

- Cortex-A53 CPU
- 64-bit Quad-Core, 1536 MHz
- ARMv8
- 2 GiB RAM
- US$ 46



ODROID-C2 devboard

Shopping List

Item	Unit cost (USD)	Number	Total cost
ODROID-C2	$ 46	20	$ 920
5V Power Supply	$ 5	20	$ 100
Micro-SD cards	$ 17	20	$ 340
LAN cables	$ 1	21	$ 21
24-port switch (TL-SG1024D)	$ 85	1	$ 85
Total			$ 1466

Rack

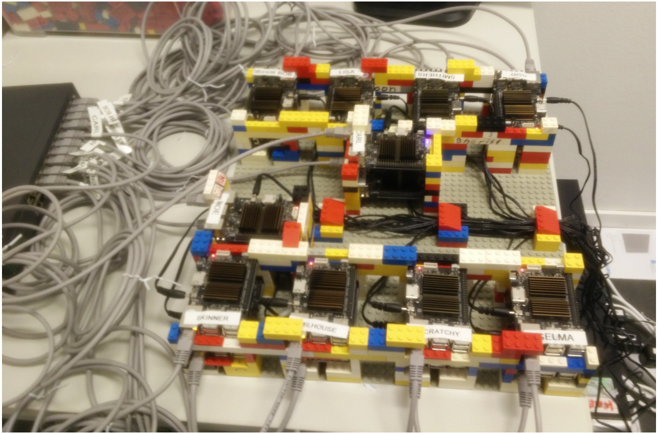


Figure: The assembled Lego “rack”. Cable management remains a subject for further investigation.

ECC2K-130

- Challenge Curves put out by Certicom in 1997 [Cer].

ECC2K-130

- Challenge Curves put out by Certicom in 1997 [Cer].
- Smaller challenges broken earlier (last 109-bit one in 2004).

ECC2K-130

- Challenge Curves put out by Certicom in 1997 [Cer].
- Smaller challenges broken earlier (last 109-bit one in 2004).
- Two curves remaining in Level I

ECC2K-130

- Challenge Curves put out by Certicom in 1997 [Cer].
- Smaller challenges broken earlier (last 109-bit one in 2004).
- Two curves remaining in Level I
  - Curve over $\mathbb{F}_p$, p a 131-bit prime

ECC2K-130

- Challenge Curves put out by Certicom in 1997 [Cer].
- Smaller challenges broken earlier (last 109-bit one in 2004).
- Two curves remaining in Level I
  - Curve over $\mathbb{F}_p$, p a 131-bit prime
  - Curve over $\mathbb{F}_{2^{131}}$, a Koblitz curve.

ECC2K-130

- Challenge Curves put out by Certicom in 1997 [Cer].
- Smaller challenges broken earlier (last 109-bit one in 2004).
- Two curves remaining in Level I
  - Curve over $\mathbb{F}_p$, p a 131-bit prime
  - Curve over $\mathbb{F}_{2^{131}}$, a Koblitz curve.
- 2009's *Breaking ECC2K-130* [Bai+09] report describes how to attack the Koblitz curve.

ECC2K-130

- Challenge Curves put out by Certicom in 1997 [Cer].
- Smaller challenges broken earlier (last 109-bit one in 2004).
- Two curves remaining in Level I
  - Curve over $\mathbb{F}_p$, p a 131-bit prime
  - Curve over $\mathbb{F}_{2^{131}}$, a Koblitz curve.
- 2009's *Breaking ECC2K-130* [Bai+09] report describes how to attack the Koblitz curve.
- The attack is based on Pollard's Rho for discrete logarithms [Pol78].

ECC2K-130

- Challenge Curves put out by Certicom in 1997 [Cer].
- Smaller challenges broken earlier (last 109-bit one in 2004).
- Two curves remaining in Level I
  - Curve over $\mathbb{F}_p$, p a 131-bit prime
  - Curve over $\mathbb{F}_{2^{131}}$, a Koblitz curve.
- 2009's *Breaking ECC2K-130* [Bai+09] report describes how to attack the Koblitz curve.
- The attack is based on Pollard's Rho for discrete logarithms [Pol78].
- They describe highly optimised implementations, speeds and estimates for CPUs, PS3s, GPUs and FPGAs.

ECC2K-130

- Challenge Curves put out by Certicom in 1997 [Cer].
- Smaller challenges broken earlier (last 109-bit one in 2004).
- Two curves remaining in Level I
  - Curve over $\mathbb{F}_p$, p a 131-bit prime
  - Curve over $\mathbb{F}_{2^{131}}$, a Koblitz curve.
- 2009's *Breaking ECC2K-130* [Bai+09] report describes how to attack the Koblitz curve.
- The attack is based on Pollard's Rho for discrete logarithms [Pol78].
- They describe highly optimised implementations, speeds and estimates for CPUs, PS3s, GPUs and FPGAs.
- To compare the ODROID-C2 to these platforms we should optimise ECC2K-130 for the Cortex-A53.

Cortex-A53 characteristics

- ARMv8-A architecture
- 32 registers
- ARM NEON extensions
  - 32 128-bit vector registers

Cortex-A53 characteristics

- ARMv8-A architecture
- 32 registers
- ARM NEON extensions
  - 32 128-bit vector registers

No detailed instruction characteristics are available

How to figure them out

- We have a cycle counter
- Idea: write small (micro) programs and measure how long they take (benchmarking).

```
measure_load:
mrs x17, PMCCNTR_EL0   ; store cycle counter at x17
ldr q0, [x0]           ; load q0 from address x0
mrs x18, PMCCNTR_EL0   ; store cycle counter at x18
sub x0, x18, x17       ; cycles spent = x18 - x19
ret
```

Benchmark results

Table: Hypothesised 128-bit vector instruction characteristics on the Cortex-A53. Latencies are including the issue cycles. `ldr` and `ldp` can be paired with a single arithmetic instruction for free.

Instruction	Issue cycles	Latency (cycles)
Binary arithmetic (`eor, and`)	1	1
Addition (`add`)	1	2
Load (`ldr`)	2	3
Store (`str`)	1	—
Load pair (`ldp`)	4	3, 4
Store pair (`stp`)	2	—

Benchmark results

Table: Hypothesised 128-bit vector instruction characteristics on the Cortex-A53. Latencies are including the issue cycles. `ldr` and `ldp` can be paired with a single arithmetic instruction for free.

Instruction	**Issue cycles**	**Latency (cycles)**
Binary arithmetic (`eor, and`)	1	1
Addition (`add`)	1	2
Load (`ldr`)	2	3
Store (`str`)	1	—
Load pair (`ldp`)	4	3, 4
Store pair (`stp`)	2	—

Execution Pipelines

```
ldr q0, [x0]
eor v1.16b, v1.16b, v1.16b
```

Instruction	Issue cycles	Latency (cycles)
Binary arithmetic (`eor, and`)	1	1
Load (`ldr`)	2	3

Bitslicing

$$
\begin{aligned}
a &= \begin{pmatrix} a_4 & a_3 & a_2 & a_1 & a_0 \end{pmatrix} \\
b &= \begin{pmatrix} b_4 & b_3 & b_2 & b_1 & b_0 \end{pmatrix} \\
c &= \begin{pmatrix} c_4 & c_3 & c_2 & c_1 & c_0 \end{pmatrix} \\
d &= \begin{pmatrix} d_4 & d_3 & d_2 & d_1 & d_0 \end{pmatrix} \\
&\vdots
\end{aligned}
$$

Bitslicing

$$\begin{pmatrix} a \\ b \\ c \\ d \\ \vdots \end{pmatrix} = \begin{pmatrix} a_4 \\ b_4 \\ c_4 \\ d_4 \\ \vdots \end{pmatrix} \begin{pmatrix} a_3 \\ b_3 \\ c_3 \\ d_3 \\ \vdots \end{pmatrix} \begin{pmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \\ \vdots \end{pmatrix} \begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ \vdots \end{pmatrix} \begin{pmatrix} a_0 \\ b_0 \\ c_0 \\ d_0 \\ \vdots \end{pmatrix}$$

Optimising n-bit binary polynomial multiplications

- Schoolbook approach: $O(n^2)$

Optimising n-bit binary polynomial multiplications

- Schoolbook approach: $O(n^2)$
- Karatsuba [KO63]: $O(n^{\log_2(3)})$

Optimising n-bit binary polynomial multiplications

- Schoolbook approach: $O(n^2)$
- Karatsuba [KO63]: $O(n^{\log_2(3)})$
  - Split A, B in an upper (A_h, B_h) and lower part (A_l, B_l)

Optimising n-bit binary polynomial multiplications

- Schoolbook approach: $O(n^2)$
- Karatsuba [KO63]: $O(n^{\log_2(3)})$
  - Split A, B in an upper (A_h, B_h) and lower part (A_l, B_l)
  - Compute $C = A \cdot B$ as
    $C = 2^n A_h \cdot B_h + 2^{n/2}(A_h + A_l) \cdot (B_h + B_l) + A_l \cdot B_l$

Optimising n-bit binary polynomial multiplications

- Schoolbook approach: $O(n^2)$
- Karatsuba [KO63]: $O(n^{\log_2(3)})$
  - Split A, B in an upper (A_h, B_h) and lower part (A_l, B_l)
  - Compute $C = A \cdot B$ as
    $C = 2^n A_h \cdot B_h + 2^{n/2}(A_h + A_l) \cdot (B_h + B_l) + A_l \cdot B_l$
- Repeat recursively

Optimising n-bit binary polynomial multiplications

- Schoolbook approach: $O(n^2)$
- Karatsuba [KO63]: $O(n^{\log_2(3)})$
  - Split A, B in an upper (A_h, B_h) and lower part (A_l, B_l)
  - Compute $C = A \cdot B$ as
    $C = 2^n A_h \cdot B_h + 2^{n/2}(A_h + A_l) \cdot (B_h + B_l) + A_l \cdot B_l$
- Repeat recursively
- You can get rid of a few operations by using Refined Karatsuba [Ber09].

Optimising n-bit binary polynomial multiplications

- Schoolbook approach: $O(n^2)$
- Karatsuba [KO63]: $O(n^{\log_2(3)})$
  - Split A, B in an upper (A_h, B_h) and lower part (A_l, B_l)
  - Compute $C = A \cdot B$ as
    $C = 2^n A_h \cdot B_h + 2^{n/2}(A_h + A_l) \cdot (B_h + B_l) + A_l \cdot B_l$
- Repeat recursively
- You can get rid of a few operations by using Refined Karatsuba [Ber09].

Optimising n-bit binary polynomial multiplications

- Schoolbook approach: $O(n^2)$
- Karatsuba [KO63]: $O(n^{\log_2(3)})$
  - Split A, B in an upper (A_h, B_h) and lower part (A_l, B_l)
  - Compute $C = A \cdot B$ as
    $C = 2^n A_h \cdot B_h + 2^{n/2}(A_h + A_l) \cdot (B_h + B_l) + A_l \cdot B_l$
- Repeat recursively
- You can get rid of a few operations by using Refined Karatsuba [Ber09].

I used Schwabe and Hutter's approach [HS15] for scheduling this in an efficient way.

Energy Usage

Item		Watts
ODROID-C2	Idle CPU load	2.3 W 5.3 W
Switch		13 W
20 ODROID-C2s	Idle CPU load	47 W 108 W
Complete System	Idle CPU load	59 W 122 W

Platform comparison

Table: ECC2K-130 on various platforms [Bai+09; Ber+10; Bos+10; Fan+10]

Type	Instance	Iters/s ($\times 10^6$)	Watts	Watts / (10^6 iters/s)
CPU	Core 2 QX6850	22.45	130 W	5.8
CPU	E5–2630L v4	61	55 W	0.9
GPU	GTX 295	63	289 W	4.6
PS3	Cell CPU	25.57	200 W	7.8
FPGA	Xilinx XC3S5000	111	5 W	0.045
ARM	ODROID-C2	3.94	5 W	1.3
	Cluster	79	122 W	1.5

Platform comparison

Table: ECC2K-130 on various platforms [Bai+09; Ber+10; Bos+10; Fan+10]

Type	Instance	Iters/s ($\times 10^6$)	Watts	Watts / (10^6 iters/s)
CPU	Core 2 QX6850	22.45	130 W	5.8
CPU	E5–2630L v4	61	55 W	0.9
GPU	GTX 295	63	289 W	4.6
PS3	Cell CPU	25.57	200 W	7.8
FPGA	Xilinx XC3S5000	111	5 W	0.045
ARM	ODROID-C2	3.94	5 W	1.3
	Cluster	79	122 W	1.5

Conclusions

- FPGAs are still the fastest choice

Conclusions

- FPGAs are still the fastest choice
  - However, they are much harder to program.

Conclusions

- FPGAs are still the fastest choice
  - However, they are much harder to program.
- Mobile CPUs are pretty good

Conclusions

- FPGAs are still the fastest choice
  - However, they are much harder to program.
- Mobile CPUs are pretty good
- More general-purpose hardware allows for easier programming

iCIS | Digital Security
Radboud University

Conclusions

- FPGAs are still the fastest choice
  - However, they are much harder to program.
- Mobile CPUs are pretty good
- More general-purpose hardware allows for easier programming
- Could also be used for teaching applications for e.g. distributed algorithms.

Conclusions

- FPGAs are still the fastest choice
  - However, they are much harder to program.
- Mobile CPUs are pretty good
- More general-purpose hardware allows for easier programming
- Could also be used for teaching applications for e.g. distributed algorithms.

Conclusions

- FPGAs are still the fastest choice
  - However, they are much harder to program.
- Mobile CPUs are pretty good
- More general-purpose hardware allows for easier programming
- Could also be used for teaching applications for e.g. distributed algorithms.

Cluster management software, benchmarking software and optimised multipliers available at thomwiggers.nl/research/armcluster/.

Conclusions

- FPGAs are still the fastest choice
  - However, they are much harder to program.
- Mobile CPUs are pretty good
- More general-purpose hardware allows for easier programming
- Could also be used for teaching applications for e.g. distributed algorithms.

Cluster management software, benchmarking software and optimised multipliers available at thomwiggers.nl/research/armcluster/.

Thank you for your attention.

Pollard's Rho

Elliptic-curve-discrete-logarithm Problem

Given points P, Q where $P = [k]Q$, find integer k.

iCIS | Digital Security
Radboud University

Pollard's Rho

Elliptic-curve-discrete-logarithm Problem

Given points P, Q where $P = [k]Q$, find integer k.

Best known attack is Pollard's Rho [Pol78]: try to find
$R = aP + bQ = a'P + b'Q$.

1. Pick a, b at random and let $R_0 = aP + bQ$.

Pollard's Rho

Elliptic-curve-discrete-logarithm Problem

Given points P, Q where $P = [k]Q$, find integer k.

Best known attack is Pollard's Rho [Pol78]: try to find $R = aP + bQ = a'P + b'Q$.

1. Pick a, b at random and let $R_0 = aP + bQ$.
2. Apply your iteration function

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

   where $j = \mathrm{HW}((x_{R_i})/2 \mod 8) + 3$.
   HW is the Hamming Weight function and σ is the Frobenius endomorphism, so $\sigma^j((x, y)) = (x^{2^j}, y^{2^j})$.

Pollard's Rho

Elliptic-curve-discrete-logarithm Problem

Given points P, Q where $P = [k]Q$, find integer k.

Best known attack is Pollard's Rho [Pol78]: try to find $R = aP + bQ = a'P + b'Q$.

1. Pick a, b at random and let $R_0 = aP + bQ$.
2. Apply your iteration function

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

   where $j = \text{HW}((x_{R_i})/2 \mod 8) + 3$.
   HW is the Hamming Weight function and σ is the Frobenius endomorphism, so $\sigma^j((x, y)) = (x^{2^j}, y^{2^j})$.
3. Repeat until you get $R_i = a'P + b'Q = R_0$ with $b \neq b'$, $k = \frac{a'-a}{b'-b}$.

iCIS | Digital Security
Radboud University

Pollard's Rho

Elliptic-curve-discrete-logarithm Problem

Given points P, Q where $P = [k]Q$, find integer k.

Best known attack is Pollard's Rho [Pol78]: try to find
$R = aP + bQ = a'P + b'Q$.

1. Pick a, b at random and let $R_0 = aP + bQ$.
2. Apply your iteration function

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

   where $j = \mathrm{HW}((x_{R_i})/2 \mod 8) + 3$.
   HW is the Hamming Weight function and σ is the Frobenius endomorphism, so $\sigma^j((x, y)) = (x^{2^j}, y^{2^j})$.
3. Repeat until you get $R_i = a'P + b'Q = R_0$ with $b \neq b'$, $k = \frac{a'-a}{b'-b}$.

Pollard's Rho

Elliptic-curve-discrete-logarithm Problem

Given points P, Q where $P = [k]Q$, find integer k.

Best known attack is Pollard's Rho [Pol78]: try to find $R = aP + bQ = a'P + b'Q$.

1. Pick a, b at random and let $R_0 = aP + bQ$.
2. Apply your iteration function

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

   where $j = \text{HW}((x_{R_i})/2 \mod 8) + 3$.
   HW is the Hamming Weight function and σ is the Frobenius endomorphism, so $\sigma^j((x, y)) = (x^{2^j}, y^{2^j})$.
3. Repeat until you get $R_i = a'P + b'Q = R_0$ with $b \neq b'$, $k = \frac{a'-a}{b'-b}$.

For ECC2K-130 an expected $2^{60.9}$ iterations are needed [Bai+09].

Distributed Pollard's Rho [OW99]

1. Do random walks on K machines.

Distributed Pollard's Rho [OW99]

1. Do random walks on K machines.
2. Have them walk until they reach a Distinguished Point.

Distributed Pollard's Rho [OW99]

1. Do random walks on K machines.
2. Have them walk until they reach a Distinguished Point.
   - In our case, when $HW(x_P) \leq 34$.

Distributed Pollard's Rho [OW99]

1. Do random walks on K machines.
2. Have them walk until they reach a Distinguished Point.
   - In our case, when $HW(x_P) \leq 34$.
3. Send the Distinguished Point (R, a, b) to the server

Distributed Pollard's Rho [OW99]

1. Do random walks on K machines.
2. Have them walk until they reach a Distinguished Point.
   - In our case, when $HW(x_P) \leq 34$.
3. Send the Distinguished Point (R, a, b) to the server
4. Server checks if has already found R with different b.

Distributed Pollard's Rho [OW99]

1. Do random walks on K machines.
2. Have them walk until they reach a Distinguished Point.
   - In our case, when $HW(x_P) \leq 34$.
3. Send the Distinguished Point (R, a, b) to the server
4. Server checks if has already found R with different b.

Distributed Pollard's Rho [OW99]

1. Do random walks on K machines.
2. Have them walk until they reach a Distinguished Point.
   - In our case, when $HW(x_P) \leq 34$.
3. Send the Distinguished Point (R, a, b) to the server
4. Server checks if has already found R with different b.

This gets us a $\Theta(K)$ speedup.

Number of operations per iteration

Iteration function
$R_{i+1} = \sigma^j(R_i) + R_i$, where $j = \mathrm{HW}((x_{R_i})/2 \mod 8) + 3$.
HW is the Hamming Weight function and σ is the Frobenius endomorphism, so $\sigma^j((x, y)) = (x^{2^j}, y^{2^j})$.

- $3 \leq j \leq 10$, so at most 20 squarings and 1 point addition.

Number of operations per iteration

Iteration function

$R_{i+1} = \sigma^j(R_i) + R_i$, where $j = \mathrm{HW}((x_{R_i})/2 \mod 8) + 3$.
HW is the Hamming Weight function and σ is the Frobenius endomorphism, so $\sigma^j((x, y)) = (x^{2^j}, y^{2^j})$.

- $3 \leq j \leq 10$, so at most 20 squarings and 1 point addition.
- In affine coordinates, this is one inversion, two multiplications, 21 squarings and seven additions over the field.

Number of operations per iteration

Iteration function

$R_{i+1} = \sigma^j(R_i) + R_i$, where $j = \text{HW}((x_{R_i})/2 \mod 8) + 3$.
HW is the Hamming Weight function and σ is the Frobenius endomorphism, so $\sigma^j((x, y)) = (x^{2^j}, y^{2^j})$.

- $3 \leq j \leq 10$, so at most 20 squarings and 1 point addition.
- In affine coordinates, this is one inversion, two multiplications, 21 squarings and seven additions over the field.
- Montgomery's trick [Mon87] allows, by batching up N inversions, to instead do $3N - 3$ more mults and only 1 inversion.

References I

ODROID-C2. Accessed 2017-04-03. URL: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G145457216438.

ARM Limited. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. 4th Sept. 2013.

References II

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 and Bo-Yin Yang. *Breaking ECC2K-130*. Cryptology ePrint Archive, Report 2009/514. 2009. URL: https://eprint.iacr.org/2009/541/.

References III

Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe and Bo-Yin Yang. 'ECC2K-130 on NVIDIA GPUs'. In: *Progress in Cryptology – INDOCRYPT 2010*. Ed. by Guang Gong and Kishan Chand Gupta. Vol. 6498. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 328–346. URL: http://cryptojedi.org/papers/#gpuev1l.

Daniel J. Bernstein. 'Batch Binary Edwards'. In: *Advances in Cryptology - CRYPTO 2009*. Ed. by Shai Halevi. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 317–336. ISBN: 978-3-642-03356-8. DOI: 10.1007/978-3-642-03356-8_19. URL: https://cr.yp.to/papers.html#bbe.

References IV

Joppe W. Bos, Thorsten Kleinjung, Ruben Niederhagen and Peter Schwabe. 'ECC2K-130 on Cell CPUs'. In: *Progress in Cryptology – AFRICACRYPT 2010*. Ed. by Daniel J. Bernstein and Tanja Lange. Vol. 6055. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 225–242. URL: http://cryptojedi.org/papers/#cbev1l.

Certicom Corp. *The Certicom ECC Challenge*. Accessed 2017-04-03. URL: https://www.certicom.com/content/certicom/en/the-certicom-ecc-challenge.html.

References V

Junfeng Fan, Daniel V. Bailey, Lejla Batina, Tim Guneysu, Christof Paar and Ingrid Verbauwhede. 'Breaking Elliptic Curve Cryptosystems Using Reconfigurable Hardware'. In: *2010 International Conference on Field Programmable Logic and Applications*. Aug. 2010, pp. 133–138. DOI: 10.1109/FPL.2010.34.

Michael Hutter and Peter Schwabe. 'Multiprecision multiplication on AVR revisited'. In: *Journal of Cryptographic Engineering* 5.3 (2015), pp. 201–214. URL: http://cryptojedi.org/papers/#avrmul.

Anatolii Karatsuba and Yu Ofman. 'Multiplication of multidigit numbers on automata'. In: *Soviet Physics Doklady*. Vol. 7. 1963, p. 595.

References VI

Peter L. Montgomery. 'Speeding the Pollard and elliptic curve methods of factorization'. In: *Mathematics of computation* 48.177 (1987), pp. 243–264.

Paul C. van Oorschot and Michael J. Wiener. 'Parallel Collision Search with Cryptanalytic Applications'. In: *Journal of Cryptology* 12.1 (1999), pp. 1–28. DOI: 10.1007/PL00003816. URL: http://dx.doi.org/10.1007/PL00003816.

John M. Pollard. 'Monte Carlo Methods for Index Computation $(\bmod p)$'. In: *Mathematics of Computation* 32.143 (1978), pp. 918–924. DOI: 10.2307/2006496. URL: http://www.jstor.org/stable/2006496.