Energy-Efficient ARM64 Cluster with Cryptanalytic Applications

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

Thom Wiggers

Institute of Computing and Information Science, Radboud University, The Netherlands
thom@thomwiggers.nl

Abstract Getting a lot of CPU power used to be an expensive undertaking. Servers with many cores cost a lot of money and consume large amounts of energy. The developments in hardware for mobile devices has resulted in a surge in relatively cheap, powerful, and low-energy CPUs. In this paper we show how to build a low-energy, eighty-core cluster built around twenty ODROID-C2 development boards for under 1500 USD. The ODROID-C2 is a 46 USD microcomputer that provides a 1.536 GHz quad-core Cortex-A53-based CPU and 2 GB of RAM. We investigate the cluster's application to cryptanalysis by implementing Pollard's Rho method to tackle the Certicom ECC2K-130 elliptic curve challenge. We optimise software from the *Breaking ECC2K-130* technical report for the Cortex-A53. To do so, we show how to use microbenchmarking to derive the needed instruction characteristics which ARM neglected to document for the public. The implementation of the ECC2K-130 attack finally allows us to compare the proposed platform to various other platforms, including "classical" desktop CPUs, GPUs and FPGAs. Although it may still be slower than for example FPGAs, our cluster still provides a lot of value for money.

Keywords: ARM, compute cluster, cryptanalysis, elliptic curve cryptography, ECC2K-130

1 Introduction

Bigger is not always better. Traditionally large computational tasks have been deployed on huge, expensive clusters. These are often comprised of a collection of energy-hungry CPUs. In recent years, GPUs, FPGAs and other accelerators have complemented these. While they provide a decent speed boost, they do not bring down the price of acquiring the hardware. It is also more difficult to write software for GPUs and FPGAs.

The rise of portable computing in smartphones, tablets and the "Internet of Things" has coincided with a surge in relatively low-cost, powerful and low-energy CPUs. We investigate a cluster built from ARM Cortex-A53 based development boards. The Cortex-A53 has been employed in many smartphones,[1] and is also the

[1] Including the Motorola Moto G5 Plus, Moto X Play, Samsung Galaxy A3 and A5, and HTC Desire 826: https://goo.gl/aZMky5

CPU powering the popular Raspberry Pi 3 [4] development board and Nintendo Switch game console [21]. We used the ODROID-C2 by Hardkernel [17]. The ODROID-C2 provides a quad-core, 1.536 GHz CPU and 2GB of RAM for 46 US Dollars.

In this paper, we will start out by showing how to build a cheap cluster from 20 ODROID-C2 boards. This includes a "shopping list" with all of the required components. We will then discuss the characteristics of the Cortex-A53. Microbenchmarking is used to determine instruction characteristics that ARM have neglected to publish.

Next, we adapted software from the international effort to break the ECC2K-130 challenge elliptic curve [3] to run on our platform. Using Karatsuba multiplication and bitslicing techniques, we are able to run 79 million Pollard Rho iterations per second on our full cluster.

Finally, the results with our ECC2K-130 software allow us to compare the cluster to several other platforms. We will compare our efforts with the implementations of ECC2K-130 on Desktop CPUs, GPUs and FPGAs. While it may still be slower than FPGAs, our cluster still provides a lot of performance on a modest budget.

2 Building a Cheap Cluster

In this section we will explain how we built the cluster. We hope this inspires people who consider building a similar setup.

We ordered hardware as per the shopping list in Table 1. The listed prices are indicative of what we paid for the components. We did however pay a bit more for the ODROID-C2s due to European availability, shipping times and taxes. The listed price is from the manufacturer at Hardkernel.com.

Table 1. Shopping list for the complete cluster. Cost of the Lego is not included.

Item	Cost per unit (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

We formatted the SD cards and prepared them with Arch Linux ARM. Ansible [1] was used to provision them with the appropriate settings and to deploy software. The provisioning scripts and Ansible playbooks are available at thomwiggers.nl/research/armcluster/.

Inspired by Cox et al.'s Raspberry Pi cluster [12], we built a Lego enclosure for the ODROID-C2 boards as seen in Figure 1. It allows us to mount the boards in such a way that we get a reasonably compact setup. It also prevents any unwanted contact between exposed metal parts of the boards (notably I/O pins). Finally, Legos allow to preserve some airflow as they are rigid enough to leave gaps between columns of the structure.

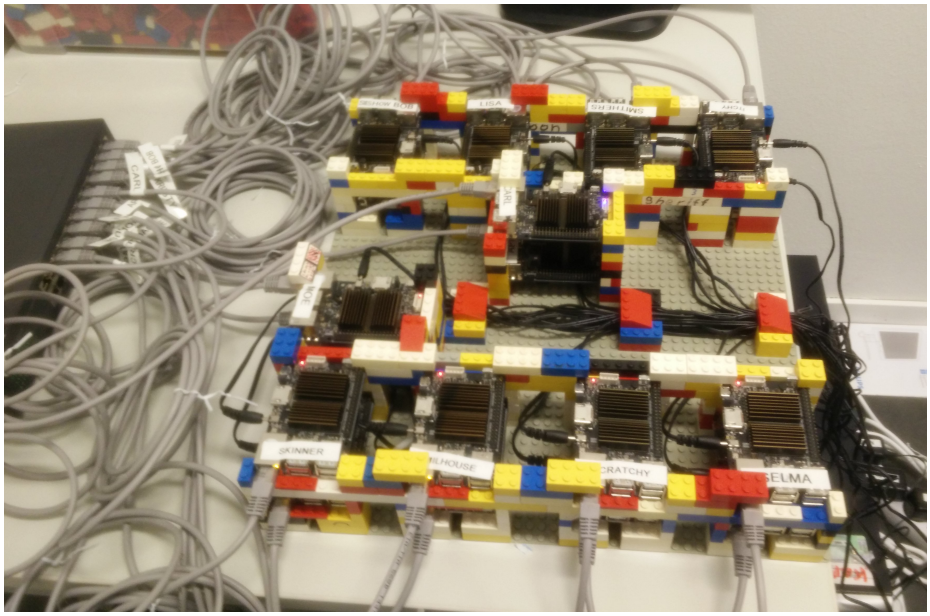

Figure 1. The assembled Lego "rack". Cable management remains a subject for further investigation.

3 The ARM Cortex-A53

The ARM Cortex-A53 is a 64-bit CPU that implements the ARMv8-A architecture. As such it provides 32 registers, twice the number of registers in the previous ARMv7 architecture. Of special interest for high-performance applications are the 128-bit NEON registers, as we will explain in Section 4.3. ARMv8 provides 32 of these SIMD registers, again doubling the number available compared to ARMv7 [2].

3.1 Determining Hardware Characteristics

To be able to understand the real potential performance of a certain CPU, we need to look at how CPU instructions behave. We need to understand not only

the number of cycles a certain instruction spends, but also the instruction latency, the delay before its result becomes available. This allows us to write software that uses the available circuits as efficiently as possible. Unfortunately, while ARM published very detailed information for previous generations of Cortex CPUs, we found information on the Cortex-A53 severely lacking. This meant we had to try and figure out cycle timings and instruction characteristics ourselves.

Fortunately, the Cortex-A53 does provide a cycle counter. It is precise enough to do benchmarking of small snippets of code. An example benchmark is given in Listing 1. This program takes nine cycles. We know that reading the cycle counter costs a cycle, so these four instructions cost eight cycles. From this we can conclude that in a sequence of loads, each load costs two cycles if they are independent.

```
fourloads: mrs x17, PMCCNTR_EL0    ; store cycle counter at x17
           ldr q0, [x0]            ; load q0 from address x0
           ldr q1, [x0]            ; load q1
           ldr q2, [x0]            ; load q2
           ldr q3, [x0]            ; load q3
           mrs x18, PMCCNTR_EL0    ; store cycle counter at x18
           sub x0, x18, x17        ; cycles spent = x18 - x19
           ret
```

Listing 1: Example microbenchmark subroutine that measures four independent 128-bit vector loads.

Listing 2 demonstrates a sequence of instructions where the second instruction uses the result of the first one. These two lines are measured to cost four cycles. We know that `ldr` is two cycles and `eor` costs a single cycle. This demonstrates that there is an extra cycle spent waiting for the result of the `ldr` to become available.

```
ldr q0, [x0]
eor v0.16b, v0.16b, v0.16b
```

Listing 2: Example microbenchmark to investigate vector load latencies. Note that `q0` and `v0` are the same register.

The two instructions shown in Listing 3 take only two cycles to execute. This is a cycle less than the simple addition of the two cycles of `ldr` and single cycle of `eor`. This shows that some instructions may execute simultaneously.

We can use these details to avoid bad schedules that cost more cycles than necessary, like that in Listing 2, and try to use those that do more in fewer cycles as in Listing 3 instead. A summary our hypotheses for 128-bit vector operations can be found in Table 2. These operations are of interest to our application,

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

Listing 3: Example microbenchmark to investigate the NEON execution pipelines.

which will become clear in Section 4. In Appendix A we will discuss some more findings for other operations and input sizes.

Detailed information on instruction characteristics is clearly of vital importance for the optimisations performed by compilers. Without it, they can hardly be expected to generate efficient instruction schedules. Unfortunately, it appears both Clang and GCC suffer from the lack of documentation. They generate arguably inefficient code in our tests with NEON programs. Consequently, to get decent performance out of the Cortex-A53 we need to hand-optimise code. We would still like to urge ARM to release better documentation for this CPU.

Table 2. Hypothesised 128-bit vector instruction characteristics on the Cortex-A53. Latencies are including the issue cycles. Vector `ldr` and `ldp` can be paired with a single arithmetic instruction for free. See Appendix A for more details.

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	—

Of special interest are the instructions that load or store two registers at the same time. When using these for vector registers, they appear to be at best as fast as the two individual instructions they would replace. "Load pair" `ldp` in particular does not appear to pair up with arithmetic instructions as well as `ldr` does, making it almost always a poor choice. Unfortunately, code generated by Clang 4.9 makes heavy use of it.

Our benchmarking software is available through our website at thomwiggers.nl/research/armcluster/. We hope that it allows others to learn more about this platform, and welcome contributions.

4 Breaking ECC on the Cortex-A53

To better understand the complexity of the elliptic-curve-discrete-logarithm problem (ECDLP) underlying elliptic curve cryptography, Certicom has published several challenges [10]. Several smaller instances have been broken already, but the 131-bit challenges, the last level-I challenges, remain open. There are two challenges over $\mathbb{F}_{2^{131}}$ and one over $\mathbb{F}_p$, where p is a 131-bit prime number.

The 2009 report "Breaking ECC2K-130" [3] describes a viable attack on the Koblitz curve challenge in $\mathbb{F}_{2^{131}}$ using Pollard's Rho method [20]. They implement their approach on various hardware platforms and give estimates how many instances of those platforms would be needed to carry out the full attack. We can build on this work by adapting the attack to our platform and providing similar estimates.

We will first explain how the attack works and analyse the expected complexity. Then we will go into adapting the attack to our target platform, the Cortex-A53. Finally, we will discuss performance estimates based on real-world benchmarks of this work.

4.1 Distributed Pollard Rho

The ECDLP problem is defined as follows: given an elliptic curve E and two points P, Q on curve E such that $Q = [k]P$, find the integer k. In other words, try to find the discrete logarithm of Q with respect to the base point P on an elliptic curve E. This problem, the basis of elliptic curve cryptography, is assumed to be hard.

Pollard's Rho algorithm for logarithms [20] is a well-known and powerful method to try to find such a k in an expected $\sqrt{\pi l/2}$ steps, where l is the order of P. By performing pseudo-random walks over the curve it tries to find integers a, b, a', b' such that $R = aP + bQ = a'P + b'Q$ with $b \neq b'$. When it finds such a solution, k can be obtained as $k = \frac{a'-a}{b'-b}$.

Van Oorschot and Wiener [18] described how to distribute this algorithm over K machines to gain a $\Theta(K)$ speedup. It works by having a server collect a subset of the points (R, a, b) computed by clients performing walks over the curve. These points are known as distinguished points. The clients submit the points with a Hamming weight of the x coordinate of R that is less than or equals 34 in a normal-basis representation. The server checks for each (R, a, b) it receives if it has already received a triple (R, a', b') where $b \neq b'$. If it has, it computes the solution k as above.

We should note that the authors of *Breaking ECC2K-130* [3] expect to break ECC2K-130 in an expected number of $2^{60.9} \in \mathcal{O}\left(\sqrt{\pi l/(2 \cdot 2 \cdot 131)}\right)$ iterations. They achieve this speedup over the general case of $\sqrt{\pi l/2}$ by applying various methods that exploit the special structure of Koblitz curves.

4.2 Iteration Function

We adopted the software from [3], which we obtained from the authors. In this section we will describe the iteration function it uses to walk over the curve.

The iteration function is defined as

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

where $j = \mathrm{HW}\left(\left(x_{R_i}\right)/2 \mod 8\right) + 3$. HW is the Hamming Weight function and σ is the Frobenius endomorphism, so $\sigma^j\left((x, y)\right) = (x^{2^j}, y^{2^j})$. This group automorphism can be used because ECC2K-130 is a Koblitz curve.

As $3 \leq j \leq 10$, putting all this together means that each iteration consists of at most twenty squarings and a single elliptic curve addition. In affine coordinates this means doing this for a single point would take one inversion, two multiplications, 21 squarings and seven additions over the underlying field. "Montgomery's trick" [16] can be used to batch up N inversions, which allows us to trade the N inversions for a single inversion and $3N - 3$ multiplications. The number of multiplications per iteration thus quickly converges to five, while the number of inversions becomes negligible for large enough N.

For the full details of the attack on ECC2K-130 we will defer to [3]. This concerns more details of the iteration function, parameters and their motivation.

4.3 Bitslicing

Bitslicing is a powerful technique used by the software from [3] that allows us to perform many operations in parallel. First we unpack the 131 bits of a $\mathbb{F}_{2^{131}}$ field element into 131 separate vectors. If we do this for many of these elements, we then can take the jth bit of all these field elements and put them all in a single vector. We do this such that in the ith vector, the jth bit represents the ith bit of the jth element. As we have 128-bit NEON vector registers on ARMv8, we store 128 bits of 128 field elements in each vector. We then use the binary logic operations on these NEON registers as if they are 128-way SIMD instructions operating on each bit simultaneously.

This technique increases the latency for a single iteration as each operation needs to be decomposed into bit-wise programs. However, because of the massive increase in parallelism we achieve a much higher throughput. Effectively, we can divide our operation counts by 128, as each bit operation manipulates 128 field elements at the same time.

4.4 Optimising Multiplications

Multiplications are the most expensive operation in terms of bit operations. Naively, they scale quadratically in the size of the input. Experimental results also show that multiplications accounted for most of the runtime of our software.

Doing schoolbook multiplication of two 131-bit polynomials would take 131^2 ANDs and 130^2 XORs, adding up to 34061 bit operations in total. Karatsuba's method [15] is a well known improvement over the classic method. We followed Hutter and Schwabe's approach [14] to efficiently schedule Karatsuba with techniques from [5, 9] to write 33- and 32-bit multipliers in assembly. We then compose these to form the full 131-bit multiplier.

Karatsuba's method computes the multiplication of two binary, n-bit polynomials by first splitting these polynomials in upper and lower parts. It then computes the product of the two upper parts, the two lower parts and the product of the addition of the upper and the lower part of the two inputs. These products are again computed using Karatsuba. Like [14] we use the refined Karatsuba approach from [5] to further reduce the number of operations needed. This comes together in Algorithm 1. Unlike the algorithms in Hutter and Schwabe's work, as

we are working on binary polynomials, we do not need to worry about carrying bits.

Algorithm 1 Refined Karatsuba, $R = A \cdot B$

Write $A \mathrel{\hat{=}} (a_0, \ldots, a_{n-1})$, $B \mathrel{\hat{=}} (b_0, \ldots, b_{n-1})$. Let $k = \frac{n}{2}$. We distinguish the upper and lower parts as $A_l \mathrel{\hat{=}} (a_0, \ldots, a_{k-1})$, $A_h \mathrel{\hat{=}} (a_k, \ldots, a_{n-1})$, $B_l \mathrel{\hat{=}} (b_0, \ldots, b_{k-1})$, $B_h \mathrel{\hat{=}} (b_k, \ldots, b_{n-1})$. The result will be given as $R \mathrel{\hat{=}} (r_0, \ldots, r_{2n-2})$.

1. Compute $A_l \cdot B_l$. Let the result be $L = (l_0, \ldots, l_{n-2})$.
2. The lower k bits of the result R are now known: $(r_0, \ldots, r_{k-1}) = (l_0, \ldots, l_{k-1})$.
3. We now compute $(A_l + A_h)$ and $(B_l + B_h)$ so we can drop the registers holding the lower parts.
4. Compute $\bar{H} = A_h \cdot B_h + (l_k, \ldots, l_{n-2}) = (\bar{h}_0, \ldots, \bar{h}_{n-2})$. It is important to do this addition during the multiplication, to minimise the number of values kept in registers.
5. Set the upper result bits: $(r_{n+k-1}, \ldots, r_{2n-2})$.
6. Compute $M = (A_l + A_h) \cdot (B_l + B_h)$.
7. Let $U = (l_0, \ldots, l_{k-1}, \bar{h}_0, \ldots, \bar{h}_{k-1})$.
8. Compute $U = U + M + \bar{H}$.
9. Set the remaining result bits $(r_k, \ldots, r_{n+k-2}) = U$.

This approach is not the cheapest in number of operations. Bernstein [5, 6] presents upper bounds for multiplication which have fewer operations. We will present a comparison in Section 5.1. However, the Bernstein multipliers are much harder to manually schedule, as they are presented as straight-line code with many intermediate values. The compilers we tried also struggled with it, and our assembly multipliers easily outperform the code generated for Bernstein's straight-line programs. These multipliers are available through our website at thomwiggers.nl/research/armcluster/.

4.5 Pollard Rho Iterations Per Second

With our high-speed multipliers, batching 32 inversions, and using bitslicing to perform 128 iterations in parallel, we achieve a speed of 1560 cycles per iteration. As the ODROID-C2 has four cores running at 1536 MHz, this means that a single board performs 3.94 million iterations per second. As we expect to need $2^{60.9}$ Pollard Rho iterations, we would need to have 17310 boards or 866 clusters running continuously to break the ECC2K-130 challenge curve in one year.

5 Results and Comparison

5.1 Benchmarking Multiplications

By using the cycle counters made available by the architecture we can obtain accurate cycle counts. We will compare our optimised assembly code with the num-

ber of bit operations and with the code given by Bernstein [6]. We compiled the code by Bernstein using Clang 4.9 with the settings `-Ofast -mtune=cortex-a53 -fomit-frame-pointer`. We will also give the number of bit operations needed by our Karatsuba approach. The published software includes our benchmarking program.

Table 3. Multiplication benchmarks. Cycle counts reported are the median of 200 000 measurements.

	4×4	8×8	16×16	32×32
Bit operations [6]	25	100	350	1158
Refined Karatsuba bit operations (Algorithm 1)	—	100	353	1168
Straight-line code [6] in C (cycles)	59	173	909	3698
This work (cycles)	42	137	416	2173

The results are shown in Table 3. The difference between the number of bit operations and the number of cycles needed can largely be found in the number of loads and stores that are needed. Some of these are required, such as the loads that are needed for the inputs ($2n$ for an $n \times n$ multiplier) and the store operations needed to write the result ($2n - 2$). Based on what we learnt from the microbenchmarks in Section 3.1 we should thus expect an additional $4n - 2$ cycles. Any further overhead results from extra spills that need to be done because not all intermediates fit into the available registers and from bad scheduling. It is not always possible to schedule the instructions in such a way that all pipeline stalls can be avoided.

The massive difference between the number of bit operations and the needed CPU cycles for the straight-line C code clearly demonstrates the poor performance of the code generated by C-compilers on the Cortex-A53. ARM would make this platform a lot more attractive if they provided the compilers with the information they need to improve instruction scheduling.

5.2 Energy Usage

We performed measurements of the energy usage of our cluster. For the measurements we used a Globaltronics GT-PM-07 Energy Meter. We measured the consumption at the wall socket, where the device or devices we were measuring were plugged in. This means that consumption includes peripheral devices like adaptors. For the measurements of multiple devices, we measured the power strip into which everything was plugged in.

Measurements were obtained for both heavy CPU load and while the cluster was idle.[2] To generate CPU load we ran our ECC2K-130 software. Results can be found in Table 4. All results have been rounded up.

Table 4. Energy Usage

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

5.3 Comparison With Other Hardware

Because the ECC2K-130 software has been optimised for many platforms, we can use it to make a comparison. The 2009 technical report [3] and the papers discussing further improvements of implementations on various platforms [7, 9, 13] provide iteration counts for various systems. While these platforms are no longer state-of-the-art, we think it still provides some insight into how our cluster compares. However, we will not compare prices of the different platforms, as some are no longer available and their retail prices are not representative anymore.

To get some more modern numbers, we adapted our software to run on AVX2 as well. In the table we list the performance of the 10-core Intel E5–2630L v4. This is a 10-core, 1.8 GHz CPU. Our implementation needs 294 cycles per iteration on this machine, for a total of 61 million iterations per second. While very fast, this CPU does come with a hefty price tag. It does illustrate that Intel CPUs also have gotten quite a bit faster.

Table 5 compares a desktop CPU, a GPU, the PlayStation 3 and a Spartan 3 FPGA with the ODROID-C2. We can see that at least per watt, the ODROID-C2 performs admirably. This conclusion is strengthened if one considers that the other platforms need more hardware. For instance, CPUs and GPUs need to be mounted on Motherboards and require additional hardware such as hard drives. These consume additional energy.

We see that FPGAs clearly outperform all competitors, including our proposal. They provide an impressive amount of iterations for very low energy. On a different curve and using a more recent FPGA, Bernstein, Engels, Lange,

[2] We should note that is important to remove the J2 jumper from the ODROID-C2 board when not powering it through USB: this saves a significant amount of energy.

Table 5. ECC2K-130 on various platforms [3, 7, 9, 13]

Type	Instance	Iters/s ($\times 10^6$)	Watts	Watts / (10^6 iters/s)	Notes
CPU	Core 2 QX6850	22.45	130 W	5.8	CPU TDP only
CPU	E5–2630L v4	61	55 W	0.9	CPU TDP only
GPU	NVIDIA GTX 295	63	289 W	4.6	GPU only
PS3	PlayStation 3 Cell CPU	25.57	200 W	7.8	Energy use while in "normal use"[19], 380 W PSU
FPGA	Xilinx XC3S5000	111	5 W	0.045	
ARM	ODROID-C2	3.94	5 W	1.3	
	ODROID-C2 Cluster	79	122 W	1.5	

Niederhagen, Paar, Schwabe and Zimmermann achieve 300 million iterations per second, although we should note that their curve is only 113 bits [8]. For purely cryptanalytic purposes FPGAs thus remain the most potent candidate.

However, our cluster is composed of more general-purpose hardware and can be used using more common programming languages. This makes it much more accessible and more generally applicable. We also see applications in education, in for example teaching distributed algorithms.

References

[1] *Ansible*. Accessed 2017-06-22. URL: https://docs.ansible.com/ansible/ (cit. on p. 2).

[2] *ARM Cortex-A Series Programmer's Guide for ARMv8-A*. Version 1.0, accessed 2017-06-22. URL: https://developer.arm.com/products/processors/cortex-a/cortex-a53/docs/den0024/latest/1-introduction (cit. on p. 3).

[3] 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/ (cit. on pp. 2, 6, 7, 10, 11).

[4] *BCM2837 – Raspberry Pi documentation*. Accessed 2017-05-08. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md (cit. on p. 2).

[5] 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 (cit. on pp. 7, 8).

[6] Daniel J. Bernstein. *Minimum number of bit operations for multiplication*. Accessed 2017-04-05. 31st May 2009. URL: https://binary.cr.yp.to/m.html (cit. on pp. 8, 9).

[7] 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 (cit. on pp. 10, 11).

[8] Daniel J. Bernstein, Susanne Engels, Tanja Lange, Ruben Niederhagen, Christof Paar, Peter Schwabe and Ralf Zimmermann. *Faster discrete logarithms on FPGAs*. 2016. URL: http://cryptojedi.org/papers/#sect113r2 (cit. on p. 11).

[9] 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 (cit. on pp. 7, 10, 11).

[10] Certicom Corp. *The Certicom ECC Challenge*. Accessed 2017-04-03. URL: https://www.certicom.com/content/certicom/en/the-certicom-ecc-challenge.html (cit. on pp. 5, 14).

[11] Certicom Research. *Certicom ECC Challenge*. 10th Nov. 2009. URL: https://www.certicom.com/content/dam/certicom/images/pdfs/challenge-2009.pdf (cit. on p. 14).

[12] Simon J. Cox, James T. Cox, Richard P. Boardman, Steven J. Johnston, Mark Scott and Neil S. O'Brien. 'Iridis-pi: a low-cost, compact demonstration cluster'. In: *Cluster Computing* 17.2 (2014), pp. 349–358. DOI: 10.1007/s10586-013-0282-7. URL: http://dx.doi.org/10.1007/s10586-013-0282-7 (cit. on p. 3).

[13] 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 (cit. on pp. 10, 11).

[14] 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 (cit. on p. 7).

[15] Anatolii Karatsuba and Yu Ofman. 'Multiplication of multidigit numbers on automata'. In: *Soviet Physics Doklady*. Vol. 7. 1963, p. 595 (cit. on p. 7).
[16] Peter L. Montgomery. 'Speeding the Pollard and elliptic curve methods of factorization'. In: *Mathematics of computation* 48.177 (1987), pp. 243–264 (cit. on p. 7).
[17] *ODROID-C2*. Accessed 2017-04-03. URL: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G145457216438 (cit. on p. 2).
[18] 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 (cit. on p. 6).
[19] Nilay Patel. *Sony says the 40GB PS3 is still using 90nm chips*. Accessed 2017-08-24. 11th Nov. 2007. URL: https://www.engadget.com/2007/11/03/sony-says-the-40gb-ps3-is-still-using-90nm-chips/ (cit. on p. 11).
[20] John M. Pollard. 'Monte Carlo Methods for Index Computation ($\mod p$)'. In: *Mathematics of Computation* 32.143 (1978), pp. 918–924. DOI: 10.2307/2006496. URL: http://www.jstor.org/stable/2006496 (cit. on p. 6).
[21] TechInsights. *Nintendo Switch teardown*. Accessed 2017-05-08. URL: http://techinsights.com/about-techinsights/overview/blog/nintendo-switch-teardown/ (cit. on p. 2).

A Cortex-A53 Benchmarking Results

In this section we will provide an overview of our results with microbenchmarking. As described in Section 3.1 we measured the execution times of various instructions. We also looked at various combinations of instructions to learn about pipelining behaviour and execution units.

A.1 Operations On "Normal" Registers

Our findings for AArch64 instructions are shown in Table 6. When measuring two arithmetic instructions we noticed that they take the time as a single instruction. This suggests that the Cortex-A53 has two ALUs and thus can compute two arithmetic instructions at the same time. This does not hold for multiplication or the memory operations and we suspect that the architecture only has one multiplier and a single processing unit for memory access.

A.2 Operations On NEON Vector Registers

The NEON vector registers are available in different sizes. It is possible to access them as 64-bit vectors or as 128-bit vectors. Tables 7 and 8 give an overview of our results. For the 64-bit vectors we again notice that two arithmetic instructions run in the same time as a single instruction. This however is not the case with

Table 6. Hypothesised instruction characteristics for instructions operating on registers. Latencies are including the issue cycles. Many instructions can be dual-issued.

Instruction	Mnemonic	Issue cycles	Latency (cycles)
Exclusive Or	`eor`	1	1
And	`and`	1	1
Or	`orr`	1	1
Or Not	`orn`	1	1
Addition	`add`	1	1
Subtraction	`sub`	1	1
Multiplication	`mul`	2	4
Load	`ldr`	1	1
Load Pair	`ldp`	2	first: 2, second: 3
Store	`str`	1	—
Store Pair	`stp`	2	—

the 128-bit vectors. This suggests that there are two 64-bit execution units that are combined for the 128-bit values.

Load and store operations again do not execute in parallel and we suspect there is only one load-store-unit. It is possible to pair up an arithmetic operation with a load or store.

B The ECC2K-130 Challenge Parameters

The Certicom ECC2K-130 challenge is defined in [10, 11]. The challenge is to find integer k such that $Q = [k]P$ on the Koblitz curve $y^2 + xy = x^3 + 1$ defined over $\mathbb{F}_{2^{131}}$. The group order $|E(\mathbb{F}_{2^{131}})| = 4l$, where l is the 129-bit prime number

$$l = 680564733841876926932320129493409985129.$$

The coordinates of P and Q are given in a polynomial-basis representation of $F_2[z]/(F)$ where $F(z) = z^{131} + z^{13} + z^2 + z + 1$. They are represented below as hexadecimal bit strings with respect to this basis.

$$\begin{aligned} P_x &= \texttt{051C99BFA6F18DE467C80C23B98C7994AA} \\ P_y &= \texttt{042EA2D112ECEC71FCF7E000D7EFC978BD} \\ Q_x &= \texttt{06C997F3E7F2C66A4A5D2FDA13756A37B1} \\ Q_y &= \texttt{04A38D11829D32D347BD0C0F584D546E9A} \end{aligned}$$

Table 7. Hypothesised instruction characteristics for instructions operating on 64-bit vectors. Latencies are including the issue cycles. Arithmetic operations can be issued together with other arithmetic instructions or with a load or store operation.

Instruction	**Mnemonic**	**Issue cycles**	**Latency (cycles)**
Exclusive Or	`eor`	1	1
And	`and`	1	1
Or	`orr`	1	1
Or Not	`orn`	1	1
Addition	`add`	1	2
Subtraction	`sub`	1	2
Multiplication	`mul`	1	4
Load	`ldr`	2	2
Load Pair	`ldp`	4	first: 3, second: 4
Store	`str`	2	—
Store Pair	`stp`	4	—

Table 8. Hypothesised instruction characteristics for instructions operating on 128-bit vectors. Latencies are including the issue cycles.

Instruction	**Mnemonic**	**Issue cycles**	**Latency (cycles)**
Exclusive Or	`eor`	1	1
And	`and`	1	1
Or	`orr`	1	1
Or Not	`orn`	1	1
Addition	`add`	1	2
Subtraction	`sub`	1	2
Multiplication	`mul`	1	4
Load	`ldr`	2	3
Load Pair	`ldp`	4	first: 4, second: 5
Store	`str`	2	—
Store Pair	`stp`	4	—