On the Deployment of curve based cryptography for the Internet of Things

Michael Scott

Cryptographic Researcher
MIRACL Labs
mscott@indigo.ie

Abstract. The typical battery supported IoT computing node has progressed in recent years from an 8-bit processor with limited memory resources, to a 32-bit processor with ample amounts of ROM and RAM. This is a game-changer for developers who no longer need to struggle with assembly language programming, but rather can bring to bear all of the tools of modern software engineering, including high level language compilers. At the same time curve based cryptography has matured to the extent that efficient curves and algorithms are now well known. However the dynamics of academic research are such that execution speed, mandating continued use of assembly language, trumps all other considerations. In this paper we report on the performance that can be expected from simple portable high-level language implementations across a wide range of contemporary architectures.
Keywords: Elliptic Curves, Pairing-based cryptography.

1 Introduction

With the advent of 5G communications there is expected to be an explosion in the Internet of Things. Yet at the moment it is widely accepted that IoT devices are often quite poorly secured. One problem we identify is that application builders have found it difficult to embed modern cryptographic protections into their small devices. There is therefore an onus on cryptographers to reduce the amount of friction involved in that process

The Arduino eco-system has long existed to provide hobbyists and researchers with a platform for experimentation. A decade ago a typical Arduino board would have been the Arduno Nano. This was based on an 8-bit Atmega chip, with up to 32KB of ROM and 2kB of RAM, and clocked at 16MHz. The latest Arduino Nano board, the Nano 33 IoT, is available in the same form-factor, but has a 32-bit ARM processor with 256KB of ROM and 32KB of RAM, clocked at 48MHz. This represents a significant paradigm shift.

Elliptic curve cryptography always held particular promise for smaller devices, requiring smaller keys and faster algorithms for group operation than legacy methods like RSA. However even with these advantages implementation on small IoT devices was challenging, and only really viable if implemented in assembly language. Often security levels would be reduced, or non-standard forms of elliptic curves would be deployed (for a recent example see [17]). A good summary would be provided in table 2 of Düll et al [12]. As can be seen such an implementation is just about possible, but can be expected to consume about half of the resources of a board like the original Arduino Nano. This leaves little room for the actual application that needs securing, as the cryptography has consumed the lions share of the boards limited resources.

It is of course a valuable and valid academic activity to leverage architectures to the maximum and to demonstrate the limits of what can be achieved. But it is less likely to lead to useful deployments of cryptography as a tool that exists in the background for securing real-world applications.

Resourceful researchers still go to heroic lengths to squeeze some meaningful cryptography into tiny spaces. And in doing do they have established an academic tradition that persists to this day, which rewards small size and fast execution, when really these are not that important anymore. In an overall design, the cryptography must not be the dominant component.

While size and speed are important, so is security against side-channel attackers, and the security that arises from confidence that the implementation is correct and does not suffer from memory leaks or other software engineering failures. Assembly language is by definition non-portable, and the IoT setting has a much wider range of competing CPU cores than the laptop/desktop/server scene where a virtual monopoly exists. Clearly it will be much harder to develop, maintain and test a collection of assembly language implementations across a multitude of different architectures.

The question therefore arises: Given the capabilities of the current generation of IoT computing nodes, what exactly can be achieved by high level language implementations? The pursuit of faster execution times to the exclusion of all else has already been questioned by Schwabe and Sprenkels [18], who point out that a slower but arguably safer elliptic curve representation may have a role to play. They also make the point that the FourQ curve [11], the undoubted current speed record holder, is in some senses seen to be unnecessarily fast.

In the same vein Aumasson [3] re-introduces us to the issue of cryptographic numerology, the unsupported and unquestioning pursuit of levels of security beyond what the science requires. The implication is that we can, in the "real world", quite safely lower our hyper-paranoid cryptographic security levels.

Of course in any discussion of elliptic curve cryptography the question of the quantum apocalypse arises. Bets have already been made on the timescale within which to expect a crypto-busting quantum computer to arise. Here we suggest an interesting side-bet: Which is more likely to be first to break a 160-bit elliptic curve, a quantum computer or a network of advanced classical computers? The implication being that if it is a quantum computer and if it happens in 20 years time, then billions of cycles would have been wasted over 20 years implementing pointlessly high levels of security.

2 Is an assembly language implementation necessary?

In the world of high-end computing, faster is always better, all other things being equal. Consider for example the case of a server farm which must handle multiple SSL connections, and where a 10% speed increase implies the need for 10% less servers, with a a 10% saving on electricity.

But in the Internet of Things, where often the Things are not under any great time pressure, we would suggest that this may be less important. But of course it depends on the actual application, and we accept that there may well be applications where it would be important in terms of the application's viability that the cryptography does not constitute a significant performance bottleneck.

The main advantages of an assembly language implementation are

- Faster execution time
- Smaller memory (ROM/RAM) requirement
- Complete control over side-channel leakage

First consider the likely execution speed advantage that can be expected by programming in assembly language. A recent paper by Alkim et al. [1], implements some post-quantum methods of cryptography based on the Ring Learning with Errors (RLWE) and related problems. By programming in assembly language and exploiting architectural features of the Cortex-M4 processor, notably its SIMD instruction set, they succeed in coding the Number Theoretic Transform which lies at the heart of such RLWE implementations, such that for a 1024-degree polynomial it executes in just 68,131 clock cycles. We find that a C language version based on [20], takes 115,698 clock cycles, about 1.7 times slower. The authors also made the rather startling observation that around 75% of the protocol's execution time was spent in hashing operations required to generate the random noise polynomials required by the protocols, and that therefore the NTT was not in fact the bottleneck calculation.

In the case of elliptic curve cryptography we find that the assembly language speed advantage can be more impressive. The popular X25519 elliptic curve [6] has been the object of much optimization and record-setting assembly language implementation over the years. In [14] a scalar multiplication can be obtained on an ARM Cortex M4 processor in just 634,567 clock cycles. We find that using C++ code the best we can do is 2,632,112 cycles, about 4 times slower.

From these examples one might have the expectation that an assembly language implementation on a 32-bit processor will be at most 2 to 4 times faster than its high level language competitor. But it would be a mistake to underestimate the ingenuity of the assembly language programmer[1]. An assembly language implementation can access architectural features like specialised SIMD instructions that are invisible to, or otherwise unusable by, a standard compiler. The assembly language programmer can also keep variables in registers for longer and minimize the number of memory accesses. In the particular case of the ARM

[1] `https://devzone.nordicsemi.com/f/nordic-q-a/18578/arm-cryptocell-310-performance`

M4 processor, they can fully exploit the powerful UMAAL instruction which in a single instruction can carry out the key operation necessary to process a partial product calculation in the context of time-critical multi-precision arithmetic. Such arithmetic lies at the heart of a prime field elliptic curve implementation.

3 High level language implementation

The advantages of writing code exclusively in a high level language, like C or C++, are

- Portability
- Maintainability
- Faster, simpler deployment. Less debugging
- More certainty around code correctness and safety

In the IoT world there are several competing architectures. The dominance of ARM is being challenged by open sourced and license-free competitors, which can result in much cheaper solutions. Amongst the alternatives we have identified are the MIPS32 and RISC-V architectures, both of which are attracting a following. Even within the ARM world there are radically different Instruction Set Architectures (ISA), from the Cortex-M0/3/4 families to the higher end models. An assembly language implementation needs to be tuned specifically for each individual ISA. This strengthens the case for a high level language solution, if one is available. It is certainly worth the effort to ascertain just what level of performance can be expected from compiler generated code.

We have accessed some representative board samples, see table 1. All the boards under consideration use a 32-bit processor. These boards are inexpensive, the cheapest being the ESP32 board at less than $5. Some allow a range of clock speeds. It is well known that the relationship between clock speed and power consumption is almost linear, so simply increasing clock speed to the maximum may not represent a viable solution. Indeed it may not produce a pro-rata improvement in execution speed as extra wait states may be inserted into each memory access, if the memory cannot keep up with the faster clocked CPU. We also indicate our very subjective view on the quality of the architecture's integer multiplication instruction, as this has a big impact on performance.

	CPU	ROM	RAM	Clock Speeds	Mul Instruction
Arduino Nano 33 IoT	ARM Cortex M0+	256K	32K	48MHz	poor
Arduino Nano 33 BLE	ARM Cortex M4	1M	256K	64MHz	excellent
Fishino Piranha	MIPS32	512K	128K	20-120MHz	good
Teensy 3.2	ARM Cortex M4	256K	64K	24-120MHz	excellent
ESP32S WROOM	Xtensa LX6	4M	512K	20-240MHz	poor
Sifive Hifive1 revb	RISC-V	4M	16K	32-320MHz	poor

Table 1. Boards tested

3.1 Is compiler generated code side-channel safe?

Before we answer this question there is an even more fundamental one. Is the processor itself side-channel safe? For example if the multiply instruction takes a variable number of cycles depending on the sizes of its operands, then it will be very difficult to defend against side-channel leakage. Ideally we need to know that the machine code instructions we will need to use, whether generated by compiler or written by hand, when they execute, do not leak information concerning their data operands. Some architectures, for example ARM Cortex-M4, are considered as OK in this regard. Others, for example ARM Cortex-M3, are not, typically because the multiply instruction takes a number of cycles that depends on the bit length of its operands. Some instructions like integer division will almost always leak information: But cryptographic code should never need integer division.

Given a good ISA, the best defense against side channel leakage is to implement all algorithms such that they execute in constant time. This should include the scenario where cache memory is used to make some memory accesses faster than others. Constant time programming is an art form in itself, but assuming that the algorithms chosen are suitable for constant time implementation, it can be achieved with careful coding, and with only a minor performance impact.

But if coding in a high-level language a problem arises: How can we be sure that the compiler has not, in the interests of optimization, translated our constant time high level code into non-constant time assembly language? Perhaps surprisingly cases have been found where this happens [13]. This is an active research area (see also [2]), but the general feeling seems to be that this is a problem which can be overcome.

As a last resort we can always visually examine the generated code and selectively turn off compiler optimization for any section of code if a non-constant time code sequence is introduced by the compiler. So we do not regard this issue as a show-stopper. In any case we now observe that many language developers are now aware of our constant time requirement, and are making efforts to help. See for example the package "subtle" in the Go programming language[2].

4 The curves

Commonly used elliptic curves in the IoT setting are the NIST standard Weierstrass secp256r1 curve, and the closely related Montgomery and Edwards curves X25519 [6] and Ed25519 [7] respectively. All provide security at the 128-bit level, which would be considered as adequate for the IoT world. However, since our high level language implementations may be slower, we will also consider a curve offering just 80-bits of security, which we call C1665. Security levels are often considered in the context of how successful cryptanalysts have been is using brute-force methods to break the hard problems that underpin security. In the case of elliptic curves over prime fields like these, this would be the discrete logarithm problem, and the current record is currently at the 56-bit level [10],

[2] `https://golang.org/pkg/crypto/subtle/`

having made little apparent progress in the last decade. C1665 is a twist-secure Edwards curves defined over the prime field $2^{166} - 5$ by the equation

$$x^2 + y^2 = 1 + 5766x^2y^2$$

The world of cryptography has had its horizons widened in the last 20 year by the advent of pairing-based cryptography which has opened up all kinds of new cryptographic possibilities, like identity-based cryptography [8], which has often been touted as having a role to play in the IoT setting. So we also consider the 254-bit "pairing-friendly" BN254 Barreto-Naehrig curve [5]. This was once assumed to exhibit 128-bits of security, but this has now been revised down to about 100-bits [15]. So we include the 381-bit BLS12-381 curve [4] which gets us back up close to the 128-bits of security.

5 Our Implementation

Surprisingly little effort has gone into the optimization of high-level language implementations of cryptography, as clearly speed records will only ever be set using assembly language. Nevertheless using some new ideas from [21], [19] incremental improvement is possible. For example the Karatsuba method for optimising multi-precision multiplication, most often requiring a careful assembly language implementation to extract its full benefit, can now easily be used from a high-level language like C++ [19].

We use our MIRACL Core multilingual library[3]. This library is written in multiple languages, which brings some advantages, as each language and compiler has its particular strengths. For example the Rust version ensures at compile time that there will be no memory leaks. By default Swift generates a run-time error on integer overflow, so it is easy to check such overflows only occur where expected. Since the implementations are straightforward transcriptions from one language to another, we can have increasing confidence that, between them, the compilers and runtimes will catch most bugs. For our tests we used the C and C++ versions of the library, as these are still the languages of choice in embedded environments. One thing that can be safely said about compilers – they can only get better. In all of our tests we use the GCC compiler, with maximum optimization of time critical sections of code. Only stack memory is used, and in some cases the default stack allocation had to be increased.

All of the devices from table 1, other than the Sifive product, are supported by the Arduino infrastructure, which makes it very easy to develop applications in C++. For the Sifive board we use the PlatformIO development tool and the C version of the MIRACL Core library.

In most cases the ROM and RAM provision was more than adequate, and our code fitted comfortably within it. The only problem was with the Sifive board, as its 16k RAM allocation was a little small for the larger pairing code. To give an indication of the ROM requirement, on the Arduino Nano 33 IoT

[3] `https://github.com/miracl/core`

board a combined implementation of the ECDH, ECDSA and ECIES protocols requires 52792 bytes of ROM memory. An implementation of the BLS signature scheme [9] on a 381-bit BLS12381 curve occupies 55192 bytes of memory. On the MIPS32 processor the same scheme occupies 89832 bytes of memory. However using MIPS16 compression, code size can be reduced by about a third, while approximately doubling execution time. In no case did the ROM requirement for our cryptographic code exceed 20% of the available allocation.

6 Results

First we attempt some comparisons with other high-level language implementations, where we could find them. In table 2 we consider the time it takes to perform a point multiplication (without precomputation) on an elliptic curve, as might arise in the context of the generation of private/public key pairs, or digital signature, or Diffie-Hellman key agreement. For comparison purposes we include some timings from the WolfSSL Benchmarks[4], and timings taken from the examples provided with the Arduino Cryptography library (ACL)[5] and the micro ECC library[6] (an * indicates an assembly language implementation). As can be seen we often achieve a useful two-times (or better) speed-up, which helps close the gap with assembly language.

	MIRACL Core		WolfSSL		ACL	μECC
Device / Elliptic Curve	secp256r1	ed25519	secp256r1	ed25519	ed25519	secp256r1
ESP32 240MHz	0.075	0.021	0.275	-	0.048	0.126
ARM M0 48MHz	1.365	0.420	3.117	-	1.033	0.825*
ARM M4 48MHz	0.170	0.056	-	-	0.241	0.187*
MIPS32 50MHz	0.197	0.070	-	-	0.177	0.370
RISC-V 320Mhz	0.151	0.024	0.550	1.436	-	-

Table 2. Elliptic curve point multiplication (in seconds)

Finally in the tables below we provide full results for all of the boards tested. Execution times are indicated in elapsed times rather than by cycle counts, as it is actual timings that ultimately matter. For standard elliptic curves we again provide timings for a scalar point multiplication. For pairing-friendly curves we time point multiplications in the elliptic curve groups $\mathbb{G}_1$ and $\mathbb{G}_2$, and exponentiation in the finite extension field $\mathbb{G}_T$, and also times for the ate pairing and the final exponentiation which calculate a mapping $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. See [16] for more details on pairings.

[4] `https://www.wolfssl.com/docs/benchmarks/`
[5] `https://github.com/rweather/arduinolibs`
[6] `https://github.com/kmackay/micro-ecc`

	Time in seconds
Curve op \ Clock Frequency	48MHz
ed25519 EC mul	0.420
secp256r1 EC mul	1.365
c1665 EC mul	0.158
bn254 $\mathbb{G}_1$ mul	0.637
bn254 $\mathbb{G}_2$ mul	1.193
bn254 $\mathbb{G}_T$ exp	1.580
bn254 pairing ate	2.028
bn254 pairing fexp	1.559
bls12381 $\mathbb{G}_1$ mul	1.175
bls12381 $\mathbb{G}_2$ mul	2.246
bls12381 $\mathbb{G}_T$ exp	2.501
bls12381 pairing ate	3.387
bls12381 pairing fexp	4.188

Table 3. Timings for Arduino Nano 33 IoT ARM Cortex-M0+ board

	Time in seconds
Curve op \ Clock Frequency	64MHz
ed25519 EC mul	0.060
secp256r1 EC mul	0.173
c1665 EC mul	0.027
bn254 $\mathbb{G}_1$ mul	0.082
bn254 $\mathbb{G}_2$ mul	0.206
bn254 $\mathbb{G}_T$ exp	0.318
bn254 pairing ate	0.339
bn254 pairing fexp	0.296
bls12381 $\mathbb{G}_1$ mul	0.141
bls12381 $\mathbb{G}_2$ mul	0.311
bls12381 $\mathbb{G}_T$ exp	0.430
bls12381 pairing ate	0.514
bls12381 pairing fexp	0.713

Table 4. Timings for Arduino Nano 33 BLE Cortex-M4 board

	Time in seconds				
Curve op \ Clock Frequency	24MHz	48MHz	72MHz	96MHz	120MHz
ed25519 EC mul	0.110	0.056	0.039	0.034	0.033
secp256r1 EC mul	0.335	0.170	0.116	0.095	0.093
c1665 EC mul	0.042	0.022	0.016	0.014	0.013
bn254 $\mathbb{G}_1$ mul	0.168	0.085	0.059	0.049	0.047
bn254 $\mathbb{G}_2$ mul	0.436	0.220	0.150	0.121	0.111
bn254 $\mathbb{G}_T$ exp	0.668	0.337	0.231	0.187	0.172
bn254 pairing ate	0.696	0.352	0.242	0.198	0.185
bn254 pairing fexp	0.609	0.308	0.212	0.174	0.162
bls12381 $\mathbb{G}_1$ mul	0.295	0.149	0.102	0.087	0.085
bls12381 $\mathbb{G}_2$ mul	0.670	0.339	0.232	0.193	0.184
bls12381 $\mathbb{G}_T$ exp	0.945	0.479	0.329	0.271	0.253
bls12381 pairing ate	1.109	0.562	0.384	0.317	0.297
bls12381 pairing fexp	1.567	0.795	0.545	0.449	0.418

Table 5. Timings for Teensy 3.2 ARM Cortex-M4 board

	Time in seconds			
Curve op \ Clock Frequency	20MHz	50MHz	80MHz	120MHz
ed25519 EC mul	0.169	0.070	0.045	0.031
secp256r1 EC mul	0.476	0.197	0.127	0.087
c1665 EC mul	0.085	0.036	0.023	0.016
bn254 $\mathbb{G}_1$ mul	0.237	0.097	0.063	0.043
bn254 $\mathbb{G}_2$ mul	0.597	0.244	0.157	0.107
bn254 $\mathbb{G}_T$ exp	0.908	0.376	0.245	0.169
bn254 pairing ate	0.959	0.396	0.258	0.178
bn254 pairing fexp	0.835	0.348	0.228	0.158
bls12381 $\mathbb{G}_1$ mul	0.426	0.173	0.110	0.075
bls12381 $\mathbb{G}_2$ mul	0.948	0.386	0.246	0.167
bls12381 $\mathbb{G}_T$ exp	1.286	0.528	0.340	0.233
bls12381 pairing ate	1.528	0.625	0.401	0.273
bls12381 pairing fexp	2.125	0.874	0.564	0.386

Table 6. Timings for Fishino Piranha MIPS32 board

	Time in seconds			
Curve op \ Clock Frequency	20MHz	40MHz	80MHz	240MHz
ed25519 EC mul	0.281	0.134	0.065	0.021
secp256r1 EC mul	0.937	0.445	0.217	0.075
c1665 EC mul	0.115	0.055	0.027	0.009
bn254 $\mathbb{G}_1$ mul	0.429	0.204	0.099	0.033
bn254 $\mathbb{G}_2$ mul	0.987	0.469	0.229	0.079
bn254 $\mathbb{G}_T$ exp	1.387	0.659	0.321	0.105
bn254 pairing ate	1.575	0.748	0.365	0.120
bn254 pairing fexp	1.305	0.619	0.302	0.099
bls12381 $\mathbb{G}_1$ mul	0.868	0.412	0.201	0.079
bls12381 $\mathbb{G}_2$ mul	1.765	0.839	0.409	0.158
bls12381 $\mathbb{G}_T$ exp	2.188	1.040	0.507	0.189
bls12381 pairing ate	2.777	1.319	0.643	0.243
bls12381 pairing fexp	3.655	1.736	0.846	0.319

Table 7. Timings for ESP32 board

	Time in seconds			
Curve op \ Clock Frequency	32MHz	64MHz	128MHz	320MHz
ed25519 EC mul	0.249	0.123	0.062	0.024
secp256r1 EC mul	1.508	0.756	0.376	0.151
c1665 EC mul	0.118	0.059	0.029	0.012
bn254 $\mathbb{G}_1$ mul	1.089	0.546	0.272	0.108
bn254 $\mathbb{G}_2$ mul	2.252	1.122	0.563	0.224
bn254 $\mathbb{G}_T$ exp	2.240	1.122	0.560	0.225
bn254 pairing ate	6.328	3.161	1.582	0.634
bn254 pairing fexp	5.740	2.864	1.438	0.572

Table 8. Timings for Sifive Hifive1 revb board

7 Conclusion

Our experiments show that even a high-level language implementation of curve-based cryptography is now viable on available IoT processor nodes, without swamping their resources. Therefore there can be no further excuses for not implementing strong security on these devices. Failure to implement such measures can now be put down to wilful negligence on the part of IoT application developers.

References

1. E. Alkim, Y. Bilgin, M. Cenk, and F. Gerárd. Cortex-M4 optimizations for {R,M}LWE schemes. Cryptology ePrint Archive, Report 2020/012, 2020. `http://eprint.iacr.org/2020/012`.
2. J. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Conference on Security Symposium*, pages 53–70, 2016.
3. J. Aumasson. Too much crypto. Cryptology ePrint Archive, Report 2019/1492, 2019. `http://eprint.iacr.org/2019/1492`.
4. P. S. L. M. Barreto, B. Lynn, and M. Scott. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks – SCN'2002*, volume 2576 of *LNCS*, pages 263–273. Springer-Verlag, 2002. `https://eprint.iacr.org/2002/088`.
5. P.S.L.M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography – SAC'2005*, volume 3897 of *LNCS*, pages 319–331, Kingston, 2006. Springer-Verlag. `https://eprint.iacr.org/2005/133`.
6. D. Bernstein. Curve25519: New Diffie-Hellman speed records. In *PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Berlin Heidelberg, 2006.

7. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Cryptology ePrint Archive, Report 2011/368, 2011. http://eprint.iacr.org/2011/368.
8. D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *SIAM Journal of Computing*, 32(3):586–615, 2003.
9. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology – Asiacrypt'2001*, volume 2248 of *LNCS*, pages 514–532. Springer-Verlag, 2002.
10. J. Bos, M. Kaihara, T. Kleinjung, A. Lenstra, and P. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *IJACT*, 2:212–228, 2012.
11. C. Costello and P. Longa. FourQ: four-dimensional decompositions on a Q-curve over the Mersenne prime. In *Asiacrypt 2015*, volume 9452 of *Lecture Notes in Computer Science*, pages 214–235, 2015.
12. M. Düll, B. Haase, G. Hinterwälder, and M. Hutter. High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 77:493–514, 2015.
13. V. Laporte G. Barthe, B. Grégoire. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *CSF 2018 - 31st IEEE Computer Security Foundations Symposium*, 2018.
14. B. Haase and B. Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. Cryptology ePrint Archive, Report 2018/286, 2018. http://eprint.iacr.org/2018/286.
15. T. Kim and R. Barbulescu. The extended tower number field sieve: A new complexity for the medium prime case. In *Crypto 2016*, volume 9814 of *LNCS*, pages 543–571. Springer-Verlag, 2016. https://eprint.iacr.org/2015/1027.
16. N. El Mrabet and M. Joye, editors. *Guide to Pairing-Based Cryptography.* Chapman and Hall/CRC, 2016. https://www.crcpress.com/Guide-to-Pairing-Based-Cryptography/El-Mrabet-Joye/p/book/9781498729505.
17. T. Pornin. Efficient elliptic curve operations on microcontrollers with finite field extensions. Cryptology ePrint Archive, Report 2020/009, 2020. http://eprint.iacr.org/2020/009.
18. P. Schwabe and D. Sprenkels. The complete cost of cofactor h=1. Cryptology ePrint Archive, Report 2019/1166, 2019. http://eprint.iacr.org/2019/1166.
19. M. Scott. Missing a trick: Karatsuba variations. Cryptology ePrint Archive, Report 2015/1247, 2015. http://eprint.iacr.org/2015/1247.
20. M. Scott. A note on the implementation of the number theoretic transform. In *IMACC 2017*, volume 10655 of *Lecture Notes in Computer Science*, pages 247–258. Springer Berlin Heidelberg, 2017.
21. M. Scott. Slothful reduction. Cryptology ePrint Archive, Report 2017/437, 2017. http://eprint.iacr.org/2017/437.