The Apache Milagro Crypto Library (V3.1)

Michael Scott

MIRACL Labs
mike.scott@miracl.com

Abstract. We describe a multi-lingual crypto library, specifically designed to support the Internet of Things.

1 Introduction

There are many crypto libraries out there. Many offer a bewildering variety of cryptographic primitives, at different levels of security. Many use extensive assembly language in order to be as fast as possible. Many are very big, even bloated. Some rely on other external libraries. Many were designed by academics for academics, and so are not really suitable for commercial use. Many are otherwise excellent, but not written in our favourite language.

The Apache Milagro Crypto Library (AMCL) [1] is different. AMCL is completely self-contained (except for the requirement for an external entropy source for random number generation). AMCL is for use in the pre-quantum era – that is in the here and now. With the advent of a workable quantum computer, AMCL will become history. But we are not expecting that to happen any time soon.

AMCL is portable – there is no assembly language. It is available in C, Java, C#, Javascript, Go, Swift and Rust using only generic programming constructs. It would be easy to develop compatible versions in other languages (see below). These versions are identical in that for the same inputs they will not only produce the same outputs, but most internal calculations will also be the same. AMCL is fast, but does not attempt to set speed records (a particular academic obsession). There are of course contexts where speed is of the essence – for example for 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 we would suggest that this is less important. In general the speed is expected to be "good enough". However AMCL is small. Some libraries boast of having hundreds of thousands of lines of code - AMCL has around 10,000. AMCL takes up the minimum of ROM/RAM resources in order to fit into the smallest possible embedded footprint, consistent with other design constraints. It is expected that this will be vital for implementations that support security in the Internet of Things. AMCL (the C version) only uses stack memory, and is thus natively multi-threaded.

The library is particularly focused on support for elliptic curve cryptography (ECC) which is gradually ousting legacy methods from their niches, and

[1] https://github.com/MIRACL/amcl.git

is particularly appropriate for the IoT. It supports all popular families of elliptic curve, at all levels of security. For legacy purposes (and processing of X.509 certificates), the RSA method is also fully supported.

AMCL makes many of the other choices for you as to which cryptographic primitives to use, based on the best available current advice. Specifically it uses AES/128/192/256 for symmetric encryption, SHA256/384/512 for hashing. It implements prime field elliptic curves for public key protocols, and BN and BLS curves to support pairing-based protocols. Three different parameterizations of Elliptic curve are supported - Weierstrass, Edwards and Montgomery, as each is appropriate within its own niche. In each case standard projective coordinates are used. But you do get to choose the actual elliptic curve, with support for three different forms of the modulus. For pairings we assume a modulus congruent to 3 mod 8 with a D-type twist [3], [2]. Standard modes of AES are supported, plus GCM mode for authenticated encryption.

The C version of AMCL is configured at compile time for 16, 32 or 64 bit processors, and for a specific elliptic curve. Interpreted languages are (obviously) processor agnostic, but the same choices of elliptic curve are available.

AMCL was written with an awareness of the abilities of modern pipelined processors. In particular there was an awareness that the unpredictable program branch should be avoided, not only as it slows down the processor, but as it may open the door to side-channel attacks. The innocuous looking `if` statement – unless its outcome can be accurately predicted – is the enemy of quality crypto software.

As is well known a first line of defence against so called side-channel attacks is to ensure as far as is possible that secret-key dependent functions run in constant time. Of course our ability to control this is dependent to an extent on the choice of language and compiler and/or virtual machine. Having a single execution path through the code, without exceptions, not only helps make the code constant time, but also makes it easier to test. We use ideas from [22] to ensure that this is the case here. To make it easier we use exception-free elliptic curve formulas where possible [18], [4], even where doing so invokes a slight degradation in performance [18].

In the sequel we refer to the C version of AMCL, unless otherwise specified. We emphasis that all AMCL versions are completely self-contained. No external libraries or packages are required to implement all of the supported cryptographic functionality (other than for an external entropy source). However we do recognise the need to support X509 standards, which while requiring no cryptographic code per se, are often required to interface closely with it. Most languages provide their own X509 packages, so we will not attempt to replicate that functionality here. However the C version of AMCL does include a basic X509 module.

2 Context

A crypto library does not function is isolation. The AMCL was originally designed to support the MIRACL IoT solution. The MIRACL IoT solution is based on a cloud-based infrastructure designed by MIRACL to support the M-Pin protocol [19], but which has wider application to novel protocols of particular relevance to the IoT. This document describes the AMCL library which was originally designed for internal use, but which has now reached a level of maturity where we are pleased to make it available as a service to the wider community as an open source product, under a standard Apache 2.0 license.

3 Library Structure

The main modules that make up AMCL are shown below, with some indication of how they interact. Several example APIs are provided to implement common protocols. Note that all interaction with the API is via machine-independent endian-indifferent arrays of bytes (a.k.a. octet strings). Therefore the inner workings of the library are invisible to the consumer of its services.

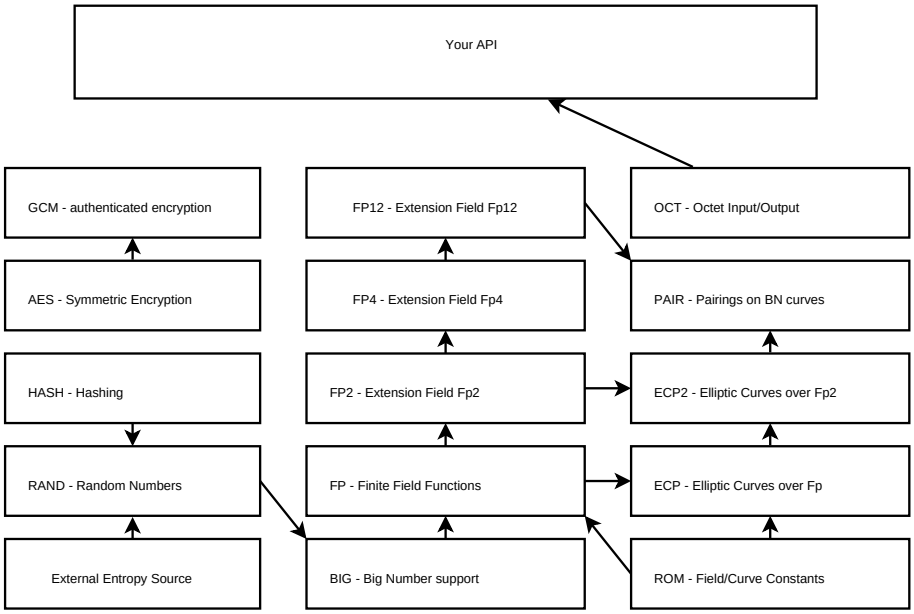

Fig. 1. The AMCL library.

The symmetric encryption and hashing code, along with the random number generation, is based on well established standards, and is not very interesting,

and since we make no claims for it, we will not refer to it again. It was mostly borrowed from our well-known MIRACL library.

4 Handling Big Numbers

4.1 Representation

One of the major design decisions is how to represent the field elements required for the elliptic curve and pairing-based cryptography. Clearly some multi-precision representation will be required. Here there are two different approaches. One is to pack the bits as tightly as possible into computer words. For example on a 64-bit computer 256-bit numbers can be stored in just 4 words. However to manipulate numbers in this form, even for simple addition, requires handling of carry bits if overflow is to be avoided, and a high-level language does not have direct access to carry flags. It is possible to emulate the flags, but this would be inefficient. In fact this approach is only really suitable for an assembly language implementation.

The alternative idea is to use extra words for the representation, and then try to offset the additional cost by taking full advantage of the "spare" bits in every word. This idea follows a "corner of the literature" [6] which has been promoted by Bernstein and his collaborators in several publications. Refer to figure 2, where a big number is represented as an array of signed integer digits, each to the base 2^b, where b is a few bits short of the full word length. Recently it has been demonstrated that such a reduced-radix representation can take advantage of a simple form of Karatsuba multiplication to significantly reduce its cost [21]. Also a reduced radix representation facilitates the implementation of constant-time modular arithmetic [22]. Such a representation of a big number is referred to as a BIG. Addition or subtraction of a pair of BIGs, results in another BIG.

Each word or limb has a "word excess" (that is the number of unused bits in every digit, excluding the sign bit). This means that multiple field elements can be added together digit by digit, without processing of carries, before overflow can occur. Only occasionally will there be a requirement to *normalise* these *extended* values, that is to force them back into the original format. Careful analysis of the code indicates where normalization is needed to avoid digit overflow. Note that this is all independent of the choice of modulus. Normally the maximum number base possible should be used. A small utility is provided with the library to assist with this choice. For example for a 32-bit processor a base of 2^{29} is almost always optimal.

Most arithmetic takes place modulo a prime number, the modulus representing the field over which the elliptic curve is defined, here denoted as p.

So as well as a "word excess", there should also be a "field excess", that is a number of extra spare bits in the most significant digit of the representation. This faciliates so-called lazy reduction (see below). We represent field elements internally as fixed length array of digits, plus an integer e which captures the worst-case excess of the element. Each field element is at all time guaranteed to be less than $e.p$.

For example for a 256-bit prime modulus on a 32-bit computer, representing field elements to the base 2^{29} allows a word excess of 2 bits and a field excess of 5 bits. The overall representation consists of 9 digits, plus a record of the current worst case excess e, so that the stored field element is less than $e.p$. To avoid overflow we must ensure that $e < 2^5$.

The other language versions can use exactly the same 32-bit representation as above. The exception is Javascript (where all numbers are stored as 64-bit floating point with a 52-bit mantissa, but mostly manipulated as 32-bit integers), where an effective word length of 26 bits or less is usually assumed.

These days many consumer products use a 64-bit processor. On these platforms this approach is even more effective, allowing much greater word and field excesses. In fact in many cases it can be proven that the excess limits will never be exceeded [22]. However one problem with 64-bit processors is the lack of language support for a 128-bit integer data type. At the time of writing only Rust and (some versions) of C support such a type. In other languages a slower workaround is used. Nevertheless 64-bit builds of the library will be faster with all languages, on a 64-bit procesor.

4.2 Addition and Subtraction

The existance of a field excess means that, independent of the word excess, multiple field elements can be added together without a requirement to immediately reduce the sum with respect to the modulus. In the literature this is referred to as lazy, or delayed, reduction. Where possible we delay a full reduction until the end of a large calculation, like an elliptic curve point multiplication [22].

Note that these two mechanisms associated with the word excess and the field excess (often confused in the literature) operate largely independently of each other.

AMCL has no support for negative numbers. However a field element x can be negated by simply calculating $-x = e.p - x$, where e is the current excess associated with x. Therefore subtraction will be implemented as field negation followed by addition. In practise it is more convenient to round e up to next highest power of 2, in which case $e.p$ can be calculated by a simple shift.

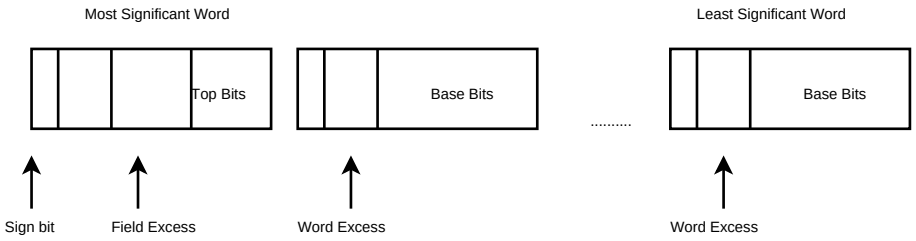

Fig. 2. Big number representation

Normalisation of extended numbers requires the word excess of each digit to be shifted right by the number of base bits, and added to the next digit, working right to left. Note that when numbers are subtracted digit-by-digit individual digits may become negative. However since we are avoiding the sign bit, due to the magic of 2's complement arithmetic, this all works fine without any conditional branches.

Final full reduction of unreduced field elements is carried out using a simple shift-and-subtract of the modulus, with one subtraction needed for every bit in the actual field excess. Such reductions will rarely be required, as they are slow and hard to do in constant time. Ideally it should only be required at the end of a complex operation like an elliptic curve point multiplication. So with careful programming we avoid any unpredictable program branches.

Since the length of field elements is fixed at compile time, it is expected that the compiler will unroll most of the time-critical loops. In any case the conditional branch required at the foot of a fixed-size loop can be accurately predicted by modern hardware.

Worst case field excesses are easy to calculate. If two elements a and b are to be added, and if their current field excesses are e_a and e_b respectively, then clearly their sum will have a worst-case field excess of $e_a + e_b$. By careful programming and choice of number base, full reductions can be largely eliminated [22].

4.3 Multiplication and Reduction

To support multiplication of BIGs, we will require a double-length DBIG type. Also the partial products that arise in the process of long multiplication will require a double-length data type. Fortunately many popular C compilers, like Gnu GCC, always support an integer type that is double the native word-length. For Java the "int" type is 32-bits and there is a double-length "long" type which is 64-bit. In common with most other langauges there is no standard 128-bit integer type. Of course for Javascript a double length type is not possible, and so the partial products must be accomodated within the 52-bit mantissa.

Multiprecision multiplication is performed column by column, propagating the carries, working from right-to-left, but using the fast method described in [21]. At the foot of each column the total is split into the sum for that column, and the carry to the next column. If the numbers are normalised prior to the multiplication, then with the word excesses that we have chosen, this will not result in overflow. The DBIG product will be automatically normalised as a result of this process. Squaring can be done in a similar fashion but at a slightly lower cost.

The method used for full reduction of a DBIG back to a BIG depends on the form of the modulus. We choose to support three distinct types of modulus, (a) pseudo Mersenne of the form $2^n - c$ where c is small and n is the size of the modulus in bits, (b) Montgomery-friendly of the form $k.2^n - 1$, and (c) moduli of no special form. For cases (b) and (c) we convert all field elements to Montgomery's n-residue form, and use Montgomery's fast method for modular reduction [16], [21]. In all cases the DBIG number to be reduced y must be

in the range $0 < y < pR$ (a requirement of Montgomery's method), and the result x is guaranteed to be in the range $0 < x < 2p$, where $R = 2^{M+FE}$ for an M-bit modulus. Note that the result will be (nearly) fully reduced. The fact than we are happy to allow x to be larger than p means that we can avoid the notorious Montgomery "final subtraction" [16]. Independent of the method used for reduction, we have found that it is much easier to obtain reduction in constant time to a value less than $2p$, than a full reduction to less than p.

Observe how unreduced numbers involved in complex calculations tend to be (nearly fully) reduced if they are involved in a modular multiplication. So for example if field element x has a large field excess, and if we calculate $x = x.y$, then as long as the unreduced product is less than pR, the result will be a nearly fully reduced x. So in many cases there is a natural tendency for field excesses not to grow without limit, and not to overflow, without requiring any explicit action on our part [22].

Consider now a sequence of code that adds, subtracts and multiplies field elements, as might arise in elliptic curve additions and doublings. Assume that the code has been analysed and that normalisation code has been inserted where needed. Assume that the reduction code that activates if there is a possibility of an element overflowing its field excess, while present, never in fact is triggered (due to the behaviour described above). Then we assert that once a program has initialised no unpredicted branches will occur during field arithmetic, and therefore the code will execute in constant time.

5 Extension Field arithmetic

To support cryptographic pairings we will need support for extension fields. We use a towering of extensions, from $\mathbb{F}_p$ to $\mathbb{F}_{p^2}$ to $\mathbb{F}_{p^4}$ to $\mathbb{F}_{p^{12}}$ as required for BN [3] and BLS [2] curves. An element of the quadratic extension field will be represented as $f = a + ib$, where i is the square root of the quadratic non-residue -1. To add, subtract and multiply them we use the obvious methods. However for negation we can construct $-f = -a - ib$ as $b - (a + b) + i.(a - (a + b)$ which requires only one base field negation. A similar idea can be used recursively for higher order extensions, so that only one base field negation is required.

6 Elliptic Curves

Three types of Elliptic curve are supported for the implementation of Elliptic Curve Cryptography (ECC), but curves are limited to popular families that support faster implementation. Weierstrass curves are supported using the Short Weierstrass representation:-

$$y^2 = x^3 + Ax + B$$

where $A = 0$ or $A = -3$. Edwards curves are supported using both regular and twisted Edwards format:-

$$Ax^2 + y^2 = 1 + Bx^2y^2$$

where $A = 1$ or $A = -1$. Montgomery curves are represented as:-

$$y^2 = x^3 + Ax^2 + x$$

where A should be small. As mentioned in the introduction, in all cases we use exception-free formulae if available, as this facilitates constant time implementation, even if this invokes a significant performance penalty [18].

For elliptic curve point multiplication, there are potentially a myriad of very dangerous side-channel attacks that arise from using the classic double-and-add algorithm and its variants. Vulnerabilities arise if branches are taken that depend on secret bits, or if data is even accessed using secret values as indices. Many types of counter-measures have been suggested. The simplest solution is to use a constant-time algorithm like the Montgomery ladder, which has a very simple structure, uses very little memory and has no key-bit-dependent branches. If using a Montgomery representation of the elliptic curve the Montgomery ladder [17] is in fact the optimal algorithm for point multiplication. For other representations we use a fixed-sized signed window method, as described in [8].

AMCL has built-in support for most standardised elliptic curves, along with many curves that have been proposed for standardisation. Specifically it supports the NIST256 curve [10], [11], the well known Curve25519 [5], the 256-bit Brainpool curve [9], the ANSSI curve [1], and six NUMS (Nothing-Up-My-Sleeve) curves proposed by Bos et al. [8]. At higher levels of security the NIST384 and NIST521 curves are supported, also Curve41417 [6] as well as the Goldilocks curve [12], and our own HiFive curve [20].

Some of these proposals support only a Weierstrass representation, but many also allow an Edwards or Montgomery form. Tools are provided to allow easy integration of more curves.

7 Support for classic Finite Field Methods

Before Elliptic Curves, cryptography depended on methods based on simple finite fields. The most famous of these would be the well known RSA method. These methods have the advantage of being effectively parameterless, and therefore the issue of trust in parameters that arises for elliptic curves, is not an issue. However these methods are subject to index calculus based methods of cryptanalysis, and so fields and keys are typically much larger. So how to support a 2048-bit implementation of RSA based on a library designed for optimized operations on much smaller numbers? The idea is simple – use AMCL as a virtual M-bit machine, where M is the bit length of the supported elliptic curve, and build RSA arithmetic on top of that. And to claw back some decent performance use the Karatsuba method [14] so that for example 2048-bit multiplication can recurse efficiently right down to 256-bit operations. Of course the downside of the Karatsuba method is that while it saves on multiplications, the number of

additions and subtractions is greatly increased. However the existance of generous word excesses in our representation makes this less of a problem, as most additions can be carried out without normalisation.

Secret key operations like RSA decryption use the Montgomery ladder to achieve side-channel-attack resistance.

The implementation can currently support $M.2^n$ bit fields. For example choosing $M = 256$, 2048-bit RSA can be used to get reasonably close to the AES-128-bit level of security, and if desired 4096 bit RSA can be used to comfortably exceed it.

Note that this code is supported independently of the elliptic curve code. So for example $M.2^n$-bit RSA and M-bit ECC can be run together within a single application.

However we regard these methods as "legacy" as in our view ECC based methods are a much better fit for the IoT.

8 Multi-Lingual support

It is a big ask to develop and maintain multiple versions of a crypto library written in radically different languages such as C, Java, Javascript, Go, Swift and Rust. This has discouraged the use of language specific methods (which are in any case of little relevance here), and strongly encouraged the use of simple, generic computer language constructs.

This approach brings a surprising bonus: AMCL can be automatically converted to many other languages using available translator tools. For example Tangible Software Solutions [23] market a Java to C# converter. This generated an efficient fully functional C# version of AMCL within minutes. The same company market a Java to Visual Basic converter. Google have a Java to Objective C converter [13] specifically designed to convert Android apps developed in Java, to iOS apps written in Objective C.

Of course not all languages can be supported in this way, and most versions were developed manually.

9 Multi-Curve support

A major innovation with version 3.0 of the library is simultaneous support for multiple curves. For example a single application might want to support both a standard elliptic curve and a pairing-friendly curve. OpenSSL would be an example of a cryptographic library that simultaneously supports multiple curves and RSA sizes. We note that Curve25519, recently added to OpenSSL, comes with its own integrated bignum library. This allows the code submitters to maintain complete control over all aspects of the implementation, right down to the lowest level. In this spirit we could see the need to extend AMCL in the same way.

The easiest way to support multiple curves, is to use distinct "namespaces", so that each curve exists completely isolated within its own namespace. And this

is the method that has been used for those languages that support namespaces. Unfortunately some languages (C, Javascript) do not, and those that do offer support, do so in rather different ways. Therefore the recommended way to build the library is now via a Python script which takes care of all these difficult details via a standardised interface. For C and Javascript we have implemented our own "namespaces" by decorating function names appropriately. For example in the 32-bit C version a BIG_256_29 type works with an FP_25519 field type to support the ECP_ED25519 Edwards curve. Similarily a BIG_256_28 type works with an FP_NIST256 field type to support the ECP_NIST256 Weierstrass curve.

10 Discussion

In our elliptic curve code we find that, by design and careful parameter choice, reductions due to possible overflow of the field excess never happen (although the code to do so is in there). Point multiplications follow exactly the same path through the code, independent of the data being processed, and hence execute in constant time. Furthermore explicit normalisation is only rarely required.

In general in developing AMCL we tried to use optimal methods, without going to what we (very subjectively) regarded as extremes in order to maximise performance. Algorithms that require less memory were generally preferred if the impact on performance was not large. Some optimizations, while perfectly valid, are hard to implement without having a significant impact on program readability and maintainability. Deciding which optimizations to use and which to reject (on the grounds of code size and negative impact on code readability and maintainability) is admittedly rather arbitrary!

One notable omission from AMCL is the use of precomputation on fixed parameters in order to speed up certain calculations. We try to justify this, rather unconvincingly, by pointing out that precomputation must of necessity increase code size. Furthermore such methods are more sensitive to side-channel attacks and much of their speed advantage will be lost if they are to be fully side-channel protected. Also precomputation on secret values clearly increases the amount of secret data that needs to be protected. Another argument against precomputation is that that which can be precomputed can, with a careful implementation, often be calculated off-line during otherwise idle processor time. However our view might change in later versions depending on our in-the-field experiences of using AMCL.

References

1. ANSSI. Publication d'un paramtrage de courbe elliptique visant des applications de passeport lectronique et de l'administration lectronique franaise., 2011. http://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000024668816.
2. 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 *Lecture Notes in Computer Science*, pages 257–267. Springer-Verlag, 2003.
3. P.S.L.M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptology – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 2006.
4. D. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Asiacrypt – 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer-Verlag, 2007.
5. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer-Verlag, 2006.
6. Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. Cryptology ePrint Archive, Report 2014/526, 2014. http://eprint.iacr.org/2014/526.
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. Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. Cryptology ePrint Archive, Report 2014/130, 2014. http://eprint.iacr.org/2014/130.
9. Brainpool. ECC brainpool standard curves and curve generation., 2005. http://www.ecc-brainpool.org/download/Domain-parameters.pdf.
10. Certicom. Sec 2: Recommended elliptic curve domain parameters, version 2.0, 2010. http://www.secg.org/download/aid-784/sec2-v2.pdf.
11. National Institute for Standards and Technology. Federal information processing standards publication 186-2, 2000. http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf.
12. M. Hamburg. Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. http://eprint.iacr.org/2015/625.
13. Google j2objc. https://github.com/google/j2objc.
14. Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.
15. A. Menezes, P. Sarkar, and S. Singh. Challenges with assessing the impact of nfs advances on the security of pairing-based cryptography. Cryptology ePrint Archive, Report 2016/1102, 2016. http://eprint.iacr.org/2016/1102.
16. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
17. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorisation. *Mathematics of Computation*, 48(177):243–264, 1987.
18. J. Renes, C. Costello, and L. Batina. Complete addition formulas for prime order elliptic curves. In *Eurocrypt – 2016*, volume 9665 of *Lecture Notes in Computer Science*, pages 403–428. Springer-Verlag, 2016.

19. M. Scott. M-Pin: A multi-factor zero knowledge authentication protocol, 2014. http://www.miracl.com/crypto-labs.
20. M. Scott. Ed3363 (highfive) – an alternative elliptic curve. Cryptology ePrint Archive, Report 2015/991, 2015. http://eprint.iacr.org/2015/991.
21. M. Scott. Missing a trick: Karatsuba variations. Cryptology ePrint Archive, Report 2015/1247, 2015. http://eprint.iacr.org/2015/1247.
22. M. Scott. Slothful reduction. Cryptology ePrint Archive, Report 2017/437, 2015. http://eprint.iacr.org/2017/437.
23. Tangible Software Solutions. http://www.tangiblesoftwaresolutions.com/.

Benchmarks

Since AMCL is intended for the Internet of Things, we think it appropriate to give some timings based on an implementation on the latest Raspberry Pi (version 3) computer, which is based on a 64-bit ARM Cortex-A53 core clocked at 1.2GHz. However although the chip is 64-bit, the Raspbian operating system is only 32-bit, so it can only run 32-bit programs

We developed three API programs, one which tests standard methods of elliptic curve key exchange, public key cryptography and digital signature. Another implements all components of our M-Pin protocol, a pairing-based protocol of medium complexity [19]. For benchmarking purposes the former was configured to use the ed25519 Edwards curve [7] with its pseudo-mersenne modulus, and the latter a 256-bit BN curve. Finally we implement all the steps of the RSA public key encryption algorithm using 2048-bit keys, that is key generation, encryption and decryption.

These might be regarded as representative of what might be expected for an implementation of a typical elliptic curve (ECC) protocol, a typical pairing-based (PBC) protocol, and a typical classic public key protocol based on RSA. The results in the first table indicate the code and stack requirements when these programs were compiled using version 4.8 of the GCC compiler, using the standard -O3 (optimize for best performance) and -Os (optimize for minimum size) flags.

	Code Size	Maximum Stack Usage
ECC -O3	68085	4140
ECC -Os	31115	3752
PBC -O3	84031	8140
PBC -Os	46044	7904
RSA -O3	61461	5332
RSA -Os	23449	5228

Table 1. Typical Memory Footprint (Raspberry Pi)

Next we give some timings for a single SPA-protected ECC point multiplication on an Edwards curve, for the calculation of a single PBC pairing on the 256-bit BN curve, and for a SPA-protected 2048-bit RSA decryption.

	Time in milliseconds
ECC point multiplication -O3	2.24
ECC point multiplication -Os	2.94
PBC pairing -O3	23.18
PBC pairing -Os	36.08
RSA decryption -O3	86.68
RSA decryption -Os	114.32

Table 2. C Benchmarks on Raspberry Pi 3

Clearly it is rather unsatisfactory to implement 32-bit programs on a 64-bit processor. So we repeated these timings on a Raspberry Pi clone, the Odroid C2, which uses the same ARM core. This board supports full 64-bit Ubuntu, and is clocked at 2GHz. The move from a 32 to 64-bit version of the library is achieved by changing a single parameter and recompiling the library.

	Time in milliseconds
ECC point multiplication -O3	0.97
ECC point multiplication -Os	1.40
PBC pairing -O3	9.01
PBC pairing -Os	14.69
RSA decryption -O3	38.46
RSA decryption -Os	63.41

Table 3. C Benchmarks on Odroid C2

Observe that we do not compare these timings with any other – because that is not the point. The point is – are they "good enough" for whatever application you have in mind? And we suspect that, in the great majority of cases, they are.

Clearly for Java and Javascript we are completely at the mercy of the efficiency (or otherwise) of the virtual machine. As can be seen from these Javascript timings, these can vary significantly.

	Device	Browser	Time in milli-seconds
ECC point multiplication	Raspberry Pi 3	Chromium	77
	Apple iPad pro	Safari	9.9
	Samsung Galaxy Note 4	Chrome	18
PBC pairing	Raspberry Pi 3	Chromium	846
	Apple iPad Pro	Safari	89
	Samsung Galaxy Note 4	Chrome	300

Table 4. JavaScript Benchmarks

(For more extensive benchmarking, run the benchmarking and testing programs now provided for each supported language.)

Addendum

Recently there has been some progress in analysis of the finite field discrete logarithm problem that applies in the context of pairing-based cryptography. As has been long suspected it is rather easier than was originally hoped. The situation is analysed thoroughly in a recent paper by Menezes, Sarkar and Singh [15]. The bottom line is that a 256-bit BN curve is no longer considered sufficient to achieve the AES-128 level of security (in fact it only achieves around 100 bits of security). A larger 384-bit BLS curve is now required to efficiently attain the originally claimed AES-128 level of security.

Therefore the AMCL library now also supports such a curve. Since this is a "larger" curve there will be an impact on performance. See tables 5, 6 and 7.

	Time in milliseconds
PBC pairing -O3	54.13
PBC pairing -Os	80.34

Table 5. C Benchmarks on Raspberry Pi 3 - 384-bit BLS curve

	Time in milliseconds
PBC pairing -O3	18.72
PBC pairing -Os	34.75

Table 6. C Benchmarks on Odroid C2 - 384-bit BLS curve

	Device	Browser	Time in milli-seconds
PBC pairing	Raspberry Pi 3	Chromium	1756
	Apple iPad Pro	Safari	186
	Samsung Galaxy Note 4	Chrome	584

Table 7. JavaScript Benchmarks - 384-bit BLS curve

Comparisons

Readers may be interested in the comparative performance of each language. Each language comes with its own benchmarking example code so that the user can check for themselves on their own hardware. Here is the output of the pairings component of the benchmark test for a 383 bit BLS pairing friendly elliptic curve for each of the directly supported languages. In each case maximal optimization is used with current versions of each language compiler/interpreter. In all cases a 64-bit build is used. The operating system is Ubuntu 16.10, running on a 1.5GHz 5 year old Asus laptop. The browser used for the Javascript is Chromium.