The MIRACL Core Library

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 MIRACL Core Library [1] is different. MIRACL Core is completely self-contained (except for the requirement for an external entropy source for random number generation). MIRACL Core is for use in the pre-quantum era – that is in the here and now. With the advent of a workable quantum computer, MIRACL Core will become history. But we are not expecting that to happen any time soon. Nonetheless MIRACL Core does include a full implementation of the NewHope post-quantum lattice-based key exchange protocol [1], which can be integrated into many cryptographic schemes right now to allow for a smooth transition to a post-quantum world.

MIRACL Core is portable – there is no assembly language. It is available in Python, C, 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. MIRACL Core 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, where often the Things are not under any great time pressure, we would suggest that this is less important. In general the speed is expected to be "good enough". However MIRACL Core is small. Some libraries boast of having hundreds of thousands of lines of code - MIRACL Core has around 10,000. MIRACL Core takes up the minimum of ROM/RAM resources in order to fit

[1] https://github.com/miracl/core

into a small 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. MIRACL Core, where allowed for by the language, only uses stack memory. It is natively multi-threaded.

To support these claims for IoT support, MIRACL Core now includes full build instructions for the Arduino IoT MKR1000 board, which is based on a 48MHz ARM Cortex M0 chip, with just 256K of ROM and 32K of RAM.

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

MIRACL Core 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 and SHA3 for hashing and message authentication (HMAC). It implements prime field elliptic curves for public key protocols, and BN and BLS curves [4], [3] 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. Standard modes of AES are supported, plus GCM mode for authenticated encryption.

The C version of MIRACL Core 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.

MIRACL Core 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 [27] to ensure that this is the case here. To make it easier we use exception-free elliptic curve formulas where possible [23], [6], even where doing so invokes a slight degradation in performance [23].

In the sequel we refer to the C version of MIRACL Core, unless otherwise specified. We emphasis that all MIRACL Core 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 MIRACL Core does include a basic X509 module.

2 Context

A crypto library does not function in isolation. MIRACL Core 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 [24], but which has wider application to novel protocols of particular relevance to the IoT. This document describes the MIRACL Core 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.

3 Library Structure

Many of the main modules that make up MIRACL Core 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.

The symmetric encryption, hashing and message authentication (HMAC) 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 large 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.

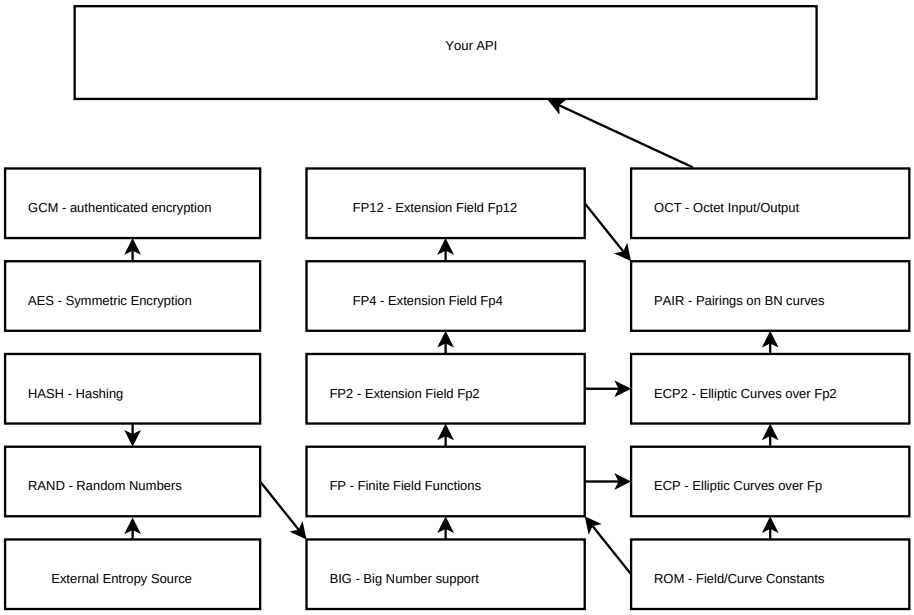


Fig. 1. The MIRACL Core library.

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" [8] 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 [26]. Also a reduced radix representation facilitates the implementation of constant-time modular arithmetic [27]. 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 [27]. 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 existence 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 [27].

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.

MIRACL Core has no support for negative numbers. **It is the programmers responsibility to make sure that a negative result can never happen. MIRACL Core does not support a general purpose big number library.** 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.

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,

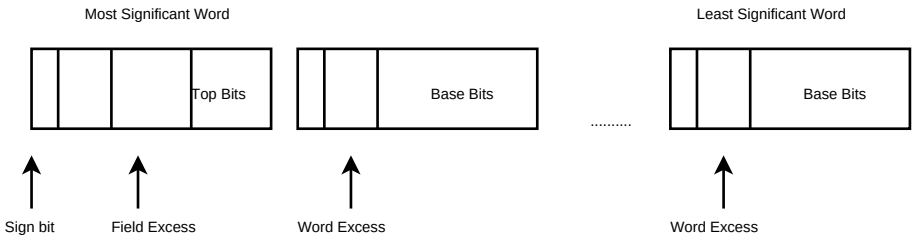


Fig. 2. Big number representation

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 [27].

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 languages 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 [26]. 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 [21], [26]. 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 and a field excess of FE. 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" [21]. 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 [27].

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 [4] curves and also from $\mathbb{F}_{p^4}$ to $\mathbb{F}_{p^8}$ to $\mathbb{F}_{p^{24}}$, and from $\mathbb{F}_{p^8}$ to $\mathbb{F}_{p^{16}}$ to $\mathbb{F}_{p^{48}}$ as required for higher security BLS [3] curves. An element of the quadratic extension field $\mathbb{F}_{p^2}$ 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 [23].

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 [22] is in fact the optimal algorithm for point multiplication. For other representations we use a fixed-sized signed window method, as described in [10].

MIRACL Core 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 [13], [15], the well known Curve25519 [7], the 256-bit Brainpool curve [11], the ANSSI curve [2], and six NUMS (Nothing-Up-My-Sleeve) curves proposed by Bos et al. [10]. At higher levels of security the NIST384 and NIST521 curves are supported, also Curve41417 [8] as well as the Goldilocks curve [17], and our own HiFive curve [25].

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 Pairings

Certain special elliptic curves support pairing based cryptography. Two type of pairing-friendly curves are supported, the BN curves [4] and the BLS curves [3]. These are so-called type-3 pairings [16]. A typical pairing-based protocol involves

point multiplications in the elliptic curve groups $\mathbb{G}_1$ and $\mathbb{G}_2$, and exponentiation in a finite extension field $\mathbb{G}_T$. The MIRACL Core library provides optimized functions for these operations, and also for the calculation of the pairing itself, which takes two inputs, one from $\mathbb{G}_1$ and the other from $\mathbb{G}_2$, with an output in $\mathbb{G}_T$. There are functions to calculate a single pairing, the product of two pairings (as often required), and a mechanism for calculating the product of many pairings – a so-called multi-pairing [28]. An example API implements the Boneh-Lynn-Shacham signature scheme [9].

8 Interfacing with the Real World

There is, in any usable cryptographic protocol, a need to map real world values like strings and other data to the complex internal mathematical structures used by the cryptography. Often a hash function is first used to create a fixed size data blob, which can then for example be mapped to a point on an elliptic curve. In fact this is a little difficult as for any given x value there might not exist such an (x, y) point. However clever methods have been discovered which do allow such a mapping, and do so in constant time (that is without opening up a side-channel for attackers). In MIRACL Core we are following standardisation developments, and such mappings can be performed deterministically, using either the Elligator 2 method [5], or the Shallue-van de Woestijne-Ulas method [12]. The actual method used depends on the type of the curve. To this end there is support for a `map2point()` function.

For those applications that do not require constant time mapping there is also a `hap2point()` function (hunt-and-peck), which simply increments x until a valid point is found.

9 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 MIRACL Core 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 [19] 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 existence 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.

10 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: MIRACL Core can be automatically converted to many other languages using available translator tools. For example Tangible Software Solutions [29] market a Java to C# converter. This generated an efficient fully functional C# version of MIRACL Core within minutes. The same company market a Java to Visual Basic converter. Google have a Java to Objective C converter [18] specifically designed to convert Android apps developed in Java, to iOS apps written in Objective C.

An exciting recent development has been WebAssembly (Wasm) [14], an assembly language that works directly in most modern browsers, and is much faster than traditional Javascript. Since MIRACL Core is written entirely in high level languages, existing tools can be used to convert versions of MIRACL Core directly into Wasm. Instructions are provided that currently allow conversion of either the Rust or C version of MIRACL Core into Wasm. Our experience is that the benefit is substantial – Wasm is up to 20 times faster than Javascript for our applications.

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

11 Multi-Curve support

A major innovation with version 3.0 of the library was 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 MIRACL Core 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.

12 Creating your own API

Although the MIRACL Core distribution contains some example APIs (for ECDH/ECCSI/ECDSA, MPIN and BLS signature), for your own project you will want to create your own bespoke API. The BLS API provides a short simple model (see for example `bls.cpp`, or `BLS.java`). Basically you need to decide on the set of functions required to implement your API. Data is passed in and out of these functions using simple generic language-independent data types (like integers and strings), and byte arrays for the larger structures. For C and C++ an octet type is provided which provides a much safer way to communicate byte arrays than via primitive "char" strings. A benefit is that there will be no "endianess" issues for the end-user to worry about. This mechanism shields the internal MIRACL Core functions from accidental misuse, and means that most MIRACL Core functions can remain "private", with only the API functions providing "public" access. It is strongly recommended that users follow this model.

13 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 MIRACL Core 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 MIRACL Core 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 in the latest version precomputation is now an option when calculating a pairing, as the pairing computation can represent a significant bottleneck. The precomputation applies when the $\mathbb{G}_2$ input is fixed. See the Boneh-Lynn-Shacham signature [9] API for an example of its use.

References

1. E. Alkim, L. Ducas, T. Poppelmann, and P. Schwabe. Post-quantum key exchange – a new hope. In *25th Usenix Security Symposium*, pages 327–343, 2016.
2. ANSSI. Publication d'un parametrage de courbe elliptique visant des applications de passeport electronique et de l'administration electronique francaise., 2011. `http://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000024668816`.
3. 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.
4. 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.
5. D. Bernstein, M. Hamburg, M. Krasnova, and T. Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC conference on computer and communications security*, pages 967–980, 2013.
6. 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.
7. 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.
8. Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. Cryptology ePrint Archive, Report 2014/526, 2014. `http://eprint.iacr.org/2014/526`.
9. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Asiacrypt 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer-Verlag, 2001.

10. 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.
11. Brainpool. ECC brainpool standard curves and curve generation., 2005. http://www.ecc-brainpool.org/download/Domain-parameters.pdf.
12. E. Brier, J. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Crypto 2010*, pages 237–254, 2010.
13. Certicom. Sec 2: Recommended elliptic curve domain parameters, version 2.0, 2010. http://www.secg.org/download/aid-784/sec2-v2.pdf.
14. A. Haas et al. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.
15. 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.
16. S. Galbraith, K. Paterson, and N. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156:3113–3121, 2008.
17. M. Hamburg. Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. http://eprint.iacr.org/2015/625.
18. Google j2objc. https://github.com/google/j2objc.
19. Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.
20. 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.
21. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
22. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorisation. *Mathematics of Computation*, 48(177):243–264, 1987.
23. 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.
24. M. Scott. M-Pin: A multi-factor zero knowledge authentication protocol, 2014. http://www.miracl.com/crypto-labs.
25. M. Scott. Ed3363 (highfive) – an alternative elliptic curve. Cryptology ePrint Archive, Report 2015/991, 2015. http://eprint.iacr.org/2015/991.
26. M. Scott. Missing a trick: Karatsuba variations. Cryptology ePrint Archive, Report 2015/1247, 2015. http://eprint.iacr.org/2015/1247.
27. M. Scott. Slothful reduction. Cryptology ePrint Archive, Report 2017/437, 2015. http://eprint.iacr.org/2017/437.
28. M. Scott. Pairing implementation revisited. Cryptology ePrint Archive, Report 2019/077, 2019. http://eprint.iacr.org/2019/077.
29. Tangible Software Solutions. http://www.tangiblesoftwaresolutions.com/.

Benchmarks

Since MIRACL Core 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 four 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 [24]. Another API implements the pairing-based Boneh-Lynn-Shacham digital signature protocol [9]. All of these APIs can be used with the full range of supported curves. Finally we implement all the steps of the RSA public key encryption algorithm for various key lengths, 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	2.91
ECC point multiplication -Os	3.85
PBC pairing -O3	28.42
PBC pairing -Os	37.33
RSA decryption -O3	60.71
RSA decryption -Os	80.83

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	1.09
ECC point multiplication -Os	1.59
PBC pairing -O3	8.61
PBC pairing -Os	13.25
RSA decryption -O3	21.00
RSA decryption -Os	26.05

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	47.15
	Apple iPad Pro	Safari	5.74
	Apple iPhone 5s	Chrome	14.45
PBC pairing	Raspberry Pi 3	Chromium	515.45
	Apple iPad Pro	Safari	67.10
	Apple iPhone 5s	Chrome	172.93

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 [20]. 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 383-bit BLS12 curve is now required to efficiently attain the originally claimed AES-128 level of security.

Therefore the MIRACL Core 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	54.66
PBC pairing -Os	74.45

Table 5. C Benchmarks on Raspberry Pi 3 - 383-bit BLS12 curve

	Time in milliseconds
PBC pairing -O3	16.80
PBC pairing -Os	25.20

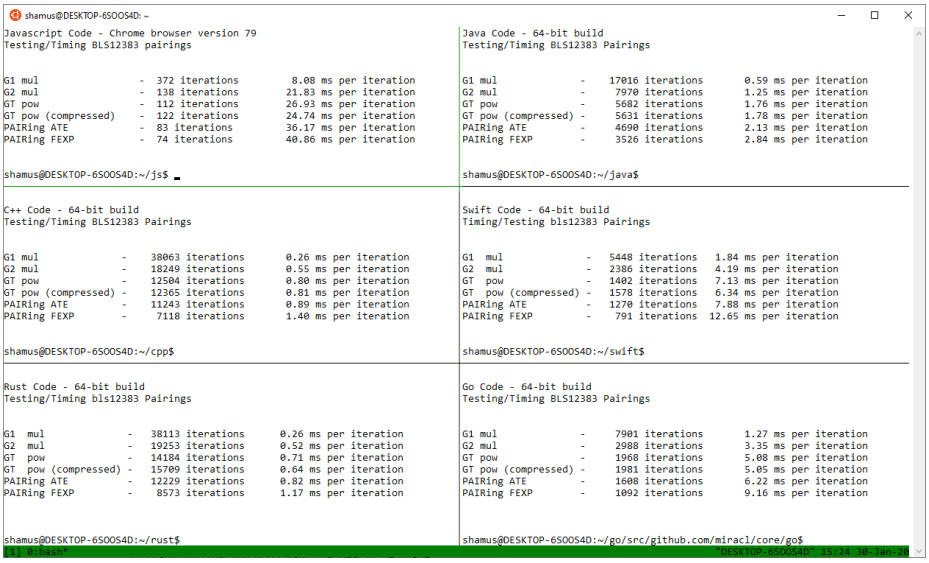
Table 6. C Benchmarks on Odroid C2 - 383-bit BLS12 curve

	Device	Browser	Time in milli-seconds
PBC pairing	Raspberry Pi 3	Chromium	1002.31
	Apple iPad Pro	Safari	151.74
	Apple iPhone 5s	Chrome	327.44

Table 7. JavaScript Benchmarks - 383-bit BLS12 curve

Comparisons

Readers may be interested in (and rather surprised by) the comparative performance of each language. For example Rust is now faster than C++. 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 BLS12 pairing friendly elliptic curve for most 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 18.04 under the Windows Subsystem for Linux, running at 3.4GHz on a i5-8250U processor.



Benchmarks for a selection of IoT boards

A case can be made that a Raspberry pi does not accurately reflect the type of computing power available to a typical Internet of Things node processor. It is after all a computer capable of running a full Linux operating system, and its clock speed is measured in Gigahertz. It requires a large battery to power it for just one day.

Therefore here we investigate the performance of the library in the context of much smaller battery powered devices. The popular low-powered boards tested are given in table 8. They are all 32-bit.

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

Table 8. Boards tested

For our applications a fast 32-bit multiply instruction is important. In this regard the ARM Cortex-M4 and MIPS32 processors are the best endowed. As a consequence, for these processors it is best to disable the Karatsuba technique as described in [26]. The ARM Cortex M0+ multiply instruction on the other hand only provides the bottom 32-bits of the product, and so is at an overall disadvantage, but therefore benefits most from using Karatsuba. This can be controlled via the USE_KARATSUBA definition in the `big.h` header file.

All of these devices, 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 idea, on the Arduino Nano 33 IoT board a combined implementation of the ECDH, ECDSA and ECIES protocols requires 52792 bytes of ROM memory. An implementation of the BLS signature scheme on a 383-bit BLS12381 curve occupies 55192 bytes of memory.

Obviously a lot depends on the compiler and the quality of the code it generates. In all cases our development tools used the well-known GCC compiler. The Arduino IDE does not always allow easy access to compiler flags, and by default optimizes for minimum size (-Os). Where possible we choose the option for maximum optimization, if one were available. In all case we inserted `#pragma`

`gcc optimize (‘‘O3’’)` at the start of the time critical `big.cpp` file, to force its compilation using maximum optimization.

Some boards allowed a range of clock speeds. Note that increasing clock speed consumes more power in a linear relationship, and may not give a pro-rata improvement in execution speed. For example extra wait states may be inserted into memory accesses, if the memory cannot keep up with the faster clocked CPU.

A variety of processors are represented, and each architecture has its own advantages and features. For example on the MIPS processor the BLS signature scheme on a 384-bit BLS12381 curve occupies 89832 bytes of memory. However using MIPS16 compression, code size can be reduced by about a third, while approximately doubling execution time. The ESP32 board is probably the cheapest, costing less than $5 per board.

In the tables below we benchmark the performance of the MIRACL Core library across the full range of IoT development boards, for typical popular curves that would be likely choices for an IoT deployment. We include a low-security curve c1665, to demonstrate what can be achieved if one is willing to compromise somewhat on cryptographic security.

There may be a concern that by only using high level languages and hence depending on compilers to generate good quality code, the performance of the MIRACL Core library may not be good enough in practice, given the complexity of the calculations. And indeed we recognize that carefully crafted assembly language implementations will always be faster, maybe up to 4 times faster. However we don't have to labour the advantages of using high level code in terms of portability and maintainability. To provide the same support, different assembly languages codes would be required for each of the five different architectures considered here.

In table 9 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. We use what are probably the most widely deployed curves, the long established standard NIST256 curve (aka secp256r1), and the more recently standardised ED25519 Edwards curve. For comparison purposes we include some timings from the WolfSSL Benchmarks, as found here[2], and timings taken from the examples provided for the Arduino Cryptography library (ACL)[3] and the micro ECC library[4] (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.

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

	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 9. Elliptic curve point multiplication (in seconds)

	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 10. 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 11. 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 12. 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 13. 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 14. 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 15. Timings for Sifive Hifive1 revb board