The MIRACL ARA Crypto Library

Michael Scott and Kealan McCusker

Certivox Labs

Abstract. We present a brief high-level description of the MIRACL ARAcrypt cryptographic library.

1 Introduction

THe MIRACL ARAcrypt library is a spin-off of our MIRACL IoT Crypto Library (MiotCL), specially tailored for the requirements of the Google ARA project. Unsurprisingly it inherits many of the features of its parent. Notably it is completely portable, but designed to be as small and as fast as possible. It is intended to be appropriate for use both in the smallest possible devices, but also on larger servers. It is written in C, but written in such a way that a translation to any other language (like Javascript, Java, Go, Swift etc.) would be quite straightforward – indeed MiotCL is already available in those languages.

The library is completely self-contained, requiring only an external entropy source which can deliver the equivalent of 256 unbiased coin tosses in order to generate any number of unguessible secret keys. The library is written with an eye on side-channel-attack resistance, to which smaller, more physically accessible, devices are potentially more vulnerable. It only uses stack memory, and is thus natively multi-threaded. The library can be configured at compile time for either 32 or 64 bit processors, and with some limitations for 16-bit processors.

The symmteric cryptographic primitives supported are standard hashing functions (SHA1/256/384/512), and AES encryption with a 128, 192, or 256 bit key. Standard modes of AES are supported, plus GCM mode for authenticated encryption.

RSA encryption is fully supported for a range of key sizes. A fixed subset of popular elliptic curves are supported at various levels of security with support for Weierstrass, Edwards and Montgomery projective parameterisation, and capable of exploiting a variety of types of modulus.

ARAcrypt is 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 if at all possible, 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.

2 Library Structure

The symmetric encryption and hashing code, along with the random number generation, is uninteresting, and since we make no special claims for it, we will not refer to it again. It was mostly borrowed directly from the MiotCL library.

3 Compile-time or run-time?

A major decision is to decide whether a library is to be designed to make the choice of RSA key sizes and/or elliptic curves at compile-time or at run-time. If decided at compile-time, then for example the elliptic curve code can be ruthlessly optimised for the one supported curve and for the one size and type of modulus. Also it can be kept really small. On the other hand if decided at run-time, clearly this allows more code re-use and flexibility, but at the cost of performance and code size (which could easily bloat to OpenSSL-like dimensions).

Here we take a hybrid approach which suits the scenario where only a finite limited number of options are required to be supported. Multiple versions of the affected modules can be provided for as many curves as are required. If only one curve needs to be supported, then all of the advantages of a compile-time decision apply. If two or three curves need to be supported then the affected code takes up two or three times the space, involving some duplication, but without impacting performance.

4 Handling Large Numbers

Another major design decision is how to represent the large field elements required for the elliptic curve cryptography. There are two popular 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. Refer to figure 1, where each digit of the representation is stored as a signed integer which is the size of the processor word-length.

Note that almost all arithmetic takes place modulo a fixed size prime number, the modulus representing the field over which an elliptic curve is defined, here denoted as p.

For example on a 32-bit processors for a 256-bit prime p, ARAcrypt represents numbers to the base 2^{29} in a 9 element array, the Word Excess is 2 bits, and the Field Excess is 5 bits.

4.1 Addition and Subtraction

The existance of a word excess means for example that multiple field elements can be added together digit by digit, without immediate 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. Note that this is independent of the modulus.

The existance of a field excess means that, independent of the word excess, multiple field elements can be added together before it is required to reduce the sum with respect to the modulus. In the literature this is referred to as lazy, or delayed, reduction.

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.

ARAcrypt has no support for negative numbers. Therefore subtraction is implemented as field negation followed by addition. Basically the number of the active bits in the field excess of the number to be negated is determined, the modulus is shifted left by this amount plus one, and the value to be negated is subtracted from this value. Note that because of the "plus 1", this will always produce a positive result at the cost of eating a bit into the field excess.

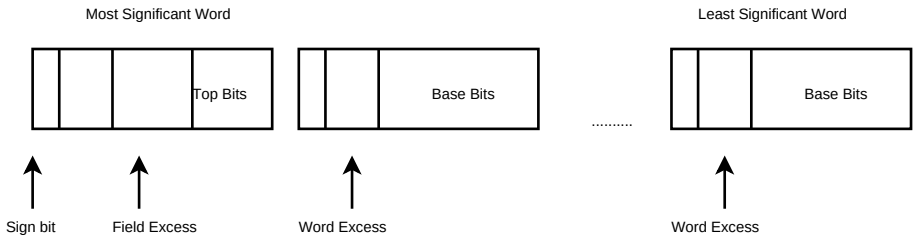

Fig. 1. BIG number representation

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 using the sign bit, due to the magic of 2's complement arithmetic, this all works fine without any conditional branches.

Reduction of unreduced BIG numbers is carried out using a simple shift-compare-and-subtract of the modulus, with one subtraction needed on average half of the time for every active bit in the field excess. Such reductions will rarely be required, as they are slow and involve 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.

The problem now is to decide when to normalise and when to reduce numbers to avoid the possibility of overflow. There are two ways of doing this. One is to monitor the excesses at run-time and act when the threat of overflow arises. The second is to do a careful analysis of the code and insert normalisation and reduction code at points where the possibility of overflow may arise, based on a static worst-case analysis.

The field excess E_n of a number n is easily captured by a simple masking and shifting of the top word. If two normalised numbers a and b are to be added then the excess of their sum will be at worst $E_a + E_b + 1$. As long as this is less than 2^{FE} where FE is the field excess, then we are fine. Otherwise both numbers should be reduced prior to the addition. In ARAcrypt these checks are performed at run-time. However, as we shall see, in practise these reductions are very rarely required. So the `if` statement used to control them is highly predictable. Observe that the field excess is always quite generous, and so many elements can be added or subtracted before reduction is required.

The worst case word excess for the result of a calculation is harder to calculate at run time, as it would require inspection of every digit of every BIG. This would slow computation down to an unacceptable extent. Therefore in this case we use static analysis and insert normalisation code where we know it might be needed. This process was supported by special debugging code that warned of places where overflow was possible, based on a simple worst-case analysis.

4.2 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.

It is generally accepted that the fastest way to do multi-precision multiplication is to accumulate the double-length partial products that contribute to each column in the classic school-boy long multiplication algorithm, also known as the Comba method. Then at the foot of the column the total is split into the sum for that column, and the carry to the next column, working right-to-left. 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 only requires just over half of the number of partial products, and so it may be somewhat faster.

The DBIG value that results from a multiplication or squaring may be immediately reduced with respect to the modulus to bring it back to a BIG. However again we may choose to delay this reduction, and therefore we need the ability to safely add and subtract DBIG numbers while again avoiding overflow.

The method used for full reduction of a DBIG back to a BIG depends on the form of the modulus. We choose to support four 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 $k.2^n - 1$, (c) generalised Mersenne of the form $2^n - 2^m - 1$, and (d) moduli of no special form. For cases (b) and (d) we convert all field elements to Montgomery's n-residue form, and use Montgomery's well known fast method for modular reduction. 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 for example $R = 2^{256+FE}$ for a 256-bit modulus. Note that the BIG result will be (nearly) fully reduced.

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 explicit action on our part.

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 there is only one possible place in which an unpredicted branch may occur. This will be in the negation code associated with a subtraction, where the number of bits in the field excess must be counted.

5 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 must be small.

In the particular case of 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 array 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 is in fact the optimal algorithm for point multiplication. For other representations we use a fixed-sized signed window method.

ARAcrypt has built-in support for standardised elliptic curves, along with curves that have been proposed for standardisation at our chosen levels of security. Specifically it supports the NIST curves with 256, 384 and 521 bit moduli, the well known Curve25519, and the 448-bit "Goldilocks" curve. Some of these support only a Weierstrass representation, but others allow an Edwards and Montgomery form. It would be easy to include more curves if required.

6 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 for example a 2048-bit implementation of RSA based on a library built primarily for optimized 256-bit operations supporting a 256-bit elliptic curve? The idea is simple – use ARAcrypt as a virtual 256-bit machine, and build 2048-bit arithmetic on top of that. And to claw back some decent performance use the Karatsuba method so that for example 2048-bit multiplication recurses 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.

The current implementation can currently support 2^n multiples of the underlying field size. Note that this code is supported independently of the elliptic curve code. So for example 2048-bit RSA and 256-bit ECC can co-exist together within the smallest applications

7 Library Structure

The library consists of the following modules, most of which have an associated header file

- `mcl_aes.c` – AES encryption
- `mcl_hash.c` – Hash functions

– mcl_gcm.c – GCM mode of AES
– mcl_rand.c – Random number generation
– mcl_oct.c – Octet Handling functions
– mcl_big.c – BIG number functions
– mcl_fp.c – BIG numbers modulo p
– mcl_ff.c – Large Finite Field functions
– mcl_ecp.c – Elliptic curve functions
– mcl_rom.c – Elliptic curve constants
– mcl_x509.c – X.509 cert processing

There are two other important header files – mcl_arch.h which defines the architecture (basically 32 or 64-bit), and mcl_config.c which configures the compile time constants for a particular elliptic curve and/or RSA key size.

As outlined above, if multiple curves and/or key sizes are to be supported, multiple versions of mcl_config.h and the mcl_big.c, mcl_fp.c, mcl_ff.c, mcl_ecp.c and mcl_rom.c are automatically generated using make tools.

Note that it is perfectly possible to create a version of the library which supports only RSA, but no elliptic curves, in which case the mcl_ecp.c, mcl_fp.c and mcl_rom.c modules can be omitted.

8 Configuration

The configuration header file mcl_config.c basically defines just 4 values.

– MCL_MBITS – the size of a BIG number in bits. Typically the size of the elliptic curve modulus.
– MCL_BASEBITS – number of "active" bits in a computer word
– MCL_MODTYPE – the type of the elliptic curve modulus
– MCL_BS – the size of a BIG in computer words. May exceed that strictly required to allow debugging.

These values are generated by makefile inputs which are selected in the file config.mk

– MCL_CHUNK – the computer word length
– MCL_CURVETYPE – the curve type
  • MCL_WEIERSTRASS
  • MCL_EDWARDS
  • MCL_MONTGOMERY
– MCL_CURVE – the curve choice
  • MCL_NIST256
  • MCL_C25519
  • MCL_C41417
  • MCL_NIST384
  • MCL_NIST521
  • MCL_C448
– MCL_FFLEN – the finite field size multiple.

The elliptic curve constants (if an elliptic curve is supported) are given in mcl_rom.c.

9 X509 Certificates

One requirement of the library is that it should be able to verify the signature on an X.509 certificate, and extract its public key along with other relevenat details, like cert ownership and validity period. One complicating factor is that a cert may be signed using one public key algorithm, while containing a public key associated with another. In theory there are a multiplicity of possible elliptic curves, finite field methods and hash functions that might appear in a valid certificate.

Clearly it would be out of the question for a library like ARAcrypt to support everything that X.509 might throw at it. Therefore it will only be expected to support a small subset.

But note that even the smallest ARAcrypt configuration could process an X.509 cert that contained an NIST256 elliptic curve public key, while being hashed using SHA256 and signed using the 2048-bit RSA public key of a certificate authority.