Universidade do Minho

Mestrado em Engenharia Informática

Tecnologia Criptográfica

Auditing a Poly1305 MAC implementation in Jasmin

Miguel Miranda Quaresma
A77049

January 1, 2019

Abstract

Poly1305 is a one time authenticator that generates a message authentication code for a given input and secret key using, for that purpose, a similar mechanism to universal hashing. Jasmin is a framework for developing high performance and high assurance cryptographic software. The present works aims to audit an implementation of the Poly1305 MAC using the Jasmin framework. I'll begin by describing the Poly1305 MAC at a high(abstraction) level, followed by an in-depth analysis of the Jasmin implementation of the algorithm. The work concludes with a formal verifcation of the assumptions that were made in the implementation.

1 Introduction

Message authentication codes play a major role in guaranteeing the authenticity and integrity of data being sent across an untrusted channel. There are many cryptoprimitives that implement MAC's and, for a long time, HMACs(Hash-MACs) were preferred over others due to their performance, with other primitives such as the ones based on universal hashing being discarded. This preference was further reinforced by the introduction of assembly instructions to perform hash functions directly in hardware [**sha˙extensions**], such as Intel's instruction for SHA-256: `sha256rnds2`. The development of cryptoprimitives such as Poly1305 MAC using Jasmin, a framework for developing high performance and high assurance cryptographic software [**jasmin˙paper**], allowed this tendency to be reversed by obtaining highly performant implementations with relative ease.

2 Poly1305 explained

Poly1305 is a message authentication code(MAC) that guarantees integrity and authenticity of messages. It achieves using a mechanism similar to a universal hash function, calculating a polynomial over a prime field to calculate a MAC for a given message,key pair. As stated Poly1305 evaluates a polynomial over a prime field, the prime, $2^{130} - 5$, being where the name (Poly**1305**) stems from. Thus, Poly1305 can be expressed by the following expression:

$$mac = (m1 * r^4 + m2 * r^3 + m3 * r^2 + m4 * r + k) \mod p$$

where mi is the i-th block of the message. Being an authenticator, Poly1305 also takes in a 256 bit secret, derived via a Password-Based Key Derivation Function(PBDKF), that is then split up into two blocks of 128 bits each, the first block being used for the parameter r which will be used as coefficient for the polynomial, and the second for the parameter k. Poly1305 works by breaking the input in 16 byte blocks, appending each block with a 1 byte(00000001) to prevent forgery, and then uses this values as the coefficients of the polynomial previously described, applying the following algorithm/formula:

```
h = 0
for block in blocks:
    h += block
    h *= r
mac = (h + k) mod 2^130-5
```

where:

- `h` is a (temporary) accumulator for the successive multiplications/additions
- `block` is a 17 byte block: 16 byte message block + 1 byte
- `r` and `k` are the 128 bit values derived from the 256 bit secret

One of the consequences of the presented algorithm is the fact that, after a certain number of blocks, the variable h will overflow due to the successive multiplications. Therefore, it is necessary to perform the calculation via modular arithmetic. To do this in an efficient manner the prime $2^{130} - 5$ was chosen to perform schoolbook multiplication in a modular fashion. To further improve performance carry propagation is delayed, made possible by the fact that a number such as $2^{130} - 5$ was chosen.

2.1 (Schoolbook) Multiplication

Following the previously presented algorithm, after adding each message block to the current h value, h is multiplied with r using schoolbook multiplication. To perform this multiplication, r and h are divided in five 4 byte(32 bit) blocks, called limbs, and multiplied with eachother as follows:

	(2^{128})	(2^{96})	(2^{64})	(2^{32})	(2^{0})
	h4	h3	h2	h1	h0
*	r4	r3	r2	r1	r0
	h4*r0	h3*r0	h2*r0	h1*r0	h0*r0
+	h3*r1	h2*r1	h1*r1	h0*r1	**5*h4*r1**
+	h2*r2	h1*r2	h0*r2	**5*h4*r2**	**5*h3*r2**
+	h1*r3	h0*r3	**5*h4*r3**	**5*h3*r3**	**5*h2*r3**
+	h0*r4	**5*h4*r4**	**5*h3*r4**	**5*h2*r4**	**5*h1*r4**

The expressions in **bold** are the modular reductions of the values overflowing 130 bits. Since the modular reduction is performed over $2^{130} - 5$, which requires 130 bits to represent, and the limbs are 32 bits, the closest multiple to 130 is 128, thus the limbs are shifted 2 bits to the right. The modular reduction is performed by multiplying the limbs by 5, due to the fact that 2^{130} is congruent with 5:

$$2^{130} \equiv 5 \pmod{2^{130} - 5}$$

hence why choosing a prime that is of the form $2^n - q$, with q being a small number(such as $2^{130} - \mathbf{5}$), is important, because it allows modular reduction to be performed by multiplying by 5 (**n.b.** this isn't a full modular reduction by $2^{130} - 5$, this one can be delayed until the very end of the MAC calculation).

2.2 Limb size

As explained before (**??**), h and r are divided in 32 bit blocks called limbs and subsequently multiplied via schoolbook multiplication. There is a caviat in this process, if a h limb spans 32 bits, a multiplication will take up 64 bits making it impossible to delay carry propagation, hindering performance significantly. To prevent this from happening, 22 bits are removed from r in the following manner:

```
r[0] = r[0] & 0x0fffffff
r[2] = r[2] & 0x0ffffffc
r[3] = r[3] & 0x0ffffffc
r[4] = r[4] & 0x0ffffffc
```

Thus each r limb will have, at most, 28 bits set (32-4). As a consequence, even if h limbs span 32 bits, a multiplication will only take up 60 bits (28+32).

After processing the entire message(**i.e** all it's blocks) the value k is added and a final full modular reduction is performed:

$$mac = (h + k) \mod 2^{130} - 5$$

the result being the MAC of the message.
The full modular reduction is just a matter of checking whether $h + k$ exceeds $2^{130} - 5$, and subtracting $2^{130} - 5$ if it is.

3 Jasmin Poly1305 Implementation

Let's now examine an implementation of Poly1305 using Jasmin. As previously stated Jasmin is a framework for developing cryptographic software inspired by qhasm, however Jasmin uses Coq proof assistant to formally verify the (assembly) code generated by the (Jasmin) compiler, providing high *assurance* high performance cryptograhic code [**jasmin˙paper**]. The function that represents the entry point for the implementation has the following signature:

```
poly1305_ref3(reg u64 out, reg u64 in, reg u64 inlen, reg u64 k)
```

This function takes as parameters `out, in` and `k` which are (64 bit) pointers to the output, input message and 256 bit secret location in memory respectively. The `inlen` parameter holds, as the name implies, the length of the input.

3.1 Setup

The parameters used in Poly1305(h, r, k) are loaded and initialized by calling

```
fn poly1305_ref3_setup(reg u64 k)
    -> reg u64[3], reg u64[2], reg u64, reg u64
```

which returns h initialized as 0, r, $r54$ and a pointer to the value of k. Using the parameter `k`, the value of r is loaded and the limb reduction($(r >> 2) * 5$) is pre calculated and stored in `r54`:

```
    r = load(k);
    r[0] &= 0x0ffffffc0fffffff;
    r[1] &= 0x0ffffffc0ffffffc;
    r54 = r[1];
    r54 >>= 2;
    r54 += r[1];
    return r, r54; // r54 = r[1] * 5/4;
```

This removes the need to perform this reduction everytime a modular operation takes place, as we'll see later.

3.2 MAC Calculation

The generation of the MAC is performed by `poly1305_ref3_update` which breaks the message in 16 byte blocks adds each block to `h` and performs the multplication with modular reduction, similar to the algorithm described in section **??**:

```
while(inlen >= 16)
  { m = load(in);
    h = add_bit(h, m, 1);
    h = mulmod(h, r, r54);
    in += 16;
    inlen -= 16;
  }
```

It first loads each 16 byte block from the memory location pointed to by `in` and adds it to the accumulator `h` by calling:

```
fn add_bit(reg u64[3] h, reg u64[2] m, inline int b)
    -> reg u64[3]
```

`add_bit` is also responsible for appending a 1 byte to each message block. The multiplication and modular reduction are then performed by calling:

```
fn mulmod(reg u64[3] h, reg u64[2] r, reg u64 r54)
   -> reg u64[3]
```

which returns the corresponding value of `h` after each iteration.

3.2.1 Modular multiplication

Before looking at the way `mulmod` performs the school book multiplication with modular reduction, it's important to note that this implementation uses limbs that are 64 bits in size(`reg u64[3] h, reg u64[2] r, reg u64 r54`) as opposed to 32 bits, hence the schoolbook multiplication takes the form of:

	(2^{128})	(2^{64})	(2^0)
	h2	h1	h0
*		r1	r0
	h2*r0	h0*r1	h0*r0
+		h1*r0	**h1*5*r1**
+		**h2*5*r1**	

which is equivalent to:

$$2^{128}*h2*r0+2^{64}*(h0*r1+h1*r0+h2*r54)+h0*r0+h1*r54$$

With this in mind we can now begin to look at the code in `mulmod` used to perform this (schoolbook) multiplication. For the first two 64 bit limbs the code that implements the calculations presented are similar. Therefore, for sake of simplicity, only the code block corresponding to $h0 * r0 + h1 * r54$ will be analyzed:

```
low = h[0];
high, low = low * r[0];
t[0] = low;
t[1] = high;
...
low = h[1];
high, low = low * r54;
cf, t[0] += low;
 _, t[1] += high + cf;
```

To perform the multiplication, two variables are declared in `mulmod`: `low` which is used to store the `h` limbs since these are used multiple times and can't be overwritten (this will not stand for the $h2 * r0$ product as we will see) and `high` which holds the carry of the product between each `r` and `h` limb. The lines

```
high, low = low * r[0];
high, low = low *r54
```

perform the products $h0 * r0$ and $h1 * (r1 >> 2) * 5$ respectively. The result of these products are then stored in the `t` array, which is seen as a little-endian number where each position represents a digit between 0 and $2^{64}-1$:

- t[0] : 2^0
- t[1] : 2^{64}
- t[2] : 2^{128}

The carry stored in `high` is added to the following 64 bit limb of `t`:

```
t[1] = high;
...
 _, t[1] += high + cf;
```

The modulo reduction is performed by multiplying $h1$ (or $h2$) with the pre calculated value $r54 = r1 >> 2 * 5$. The same mechanism is used for $h2 * 5 * r1$:

```
low = h[2];
low *= r54;
```

After performing all the smaller(magnitude) operations, $h2 * r0$ is performed inplace: `h[2] *= r[0];` and the value of `h` is merged with that of `t`. Since `h[2]` is a 64 bit value, as are `h[1]` and `h[0]`, we need to reduce `h[2]` to make sure h has, at most, 130 bits set. First we we shift the value of `h[2]` 2 bits to the right keeping the 2 least significant bits to make it 130 bits since `h[1]` and `h[0]` already have 64 bits each, totalling 128 bits. After the shift we use the modular reduction trick by multiplying $h[2] >> 2$ with 5:

```
h2r = h[2];
h2rx4 = h[2];
h[2] &= 3; // clear the remaining bits
h2r >>= 2; // (h[2]>>2)
h2rx4 &= -4; // clear first 2 bits: (h[2]>>2)<<2
h2r += h2rx4;
```

The value in $h2r$ ($h2r$educed) is then added to h[0] and the carry is propagated:

```
cf, h[0] += h2r;
cf, h[1] += 0 + cf;
_, h[2] += 0 + cf;
```

making `h` $= (h * r) \mod 2^{130}$, which is returned by `mulmod`.

After all the 16 byte blocks have been digested, `poly1305_ref3_last` handles the remaining bytes, when the message size isn't a multiple of 16 bytes:

```
if(inlen > 0)
  { m = load_last(in, inlen);
    // load last already sets the last bit
    h = add_bit(h, m, 0);
    h = mulmod(h, r, r54);
  }
```

In this case, `load_last` is responsible for setting the last bit of the block to one:

```
  s[0] = 0;
  s[1] = 0;

  j = 0;
  while(j < len)
  { c = (u8)[ptr + j];
    s[u8 (int)j] = c;
    j += 1;
  }

  s[u8 (int)j] = 0x1; //sets last byte to zero
```

3.2.2 Full modular reduction

The full modular reduction, as we already mentioned, is only performed at the end by calling:

```
fn freeze(reg u64[3] h) -> reg u64[3]
```

4 Code verification/audition

Formal verification is one of the most important parts of developing software systems used in crytical environments as it provides mathematical proof of the correctness of an implementation without needing to test all the possible inputs/use cases. There are many tools available to perform formal verification of cryptographic and other software systems(see [**coq˙proof**] and [**easycrypt**]) however the proof I'll present here serves only as a possible template from which to build proofs with such tools. It's important to take into account the fact that this assumptions were made to allow for a faster implementation. A close inspection of the code allows us to identify that the assumptions made throught the development all in the form of comments, mainly on the `mulmod` function, as the developer points out:

```
// note: throughout this function there are
// some --informal-- comments regarding
// safety. The main goal is to count the
// maximum number of bits that are needed
// at each point. TODO: formally verify
//the notes (if necessary for the safety
// analysis)
//
fn mulmod(reg u64[3] h, reg u64[2] r, reg u64 r54)
-> reg u64[3]
```

as such we'll verify each of this assumptions individually.

5 Conclusion

Futher work, develop Coq proof assistant proof of the implementation.