Universidade do Minho

Mestrado em Engenharia Informática

Tecnologia Criptográfica

Auditing a Poly1305 MAC implementation in Jasmin

Miguel Miranda Quaresma
A77049

December 29, 2018

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 [2], 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 [1], 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 so similarly to a universal hash function, using 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 k, and the second for the parameter r. Poly1305 works by breaking the input in 16 byte blocks, appending each block with a 1 byte(00000001) to prevent forgery. It then proceeds to apply 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 16 byte message block plus a 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 futhermore, carry propagation is delayed to allow for fast(er) modular reduction.

2.1 (Schoolbook) Multiplication

Following the previously presented the 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	**4*h4*r1**
+	h2*r2	h1*r2	h0*r2	**4*h4*r2**	**4*h3*r2**
+	h1*r3	h0*r3	**4*h4*r3**	**4*h3*r3**	**4*h2*r3**
+	h0*r4	**4*h4*r4**	**4*h3*r4**	**4*h2*r4**	**4*h1*r4**

The expressions in bold are the modular reductions of the values overflowing 128 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 then possible 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 (2.1), 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 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:

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

and the result is 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. 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 [1]. 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 in 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 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` and, when the message size isn't a multiple of 16 bytes, the remaining bytes are processed by `poly1305_ref3_last`.
`poly1305_ref3_update` works applying the algorithm described in section 2. It first loads each message (16 byte) block from memory location pointed to by `in` pointer, and adds it to the accumulator `h` by calling `add_bit`. `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/message block.

3.2.1 Modular multiplication

One of the things to note in the implementation being examined is the fact that limbs are 64 bits in size, 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**	

This format is can also be presented as:

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

and represents a slight variation over the previously multiplication using 32 bit limbs.

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 limbs of 64 bits 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 explained.

```
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 this 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 result of the products is then stored in the `t` array, which is seen as a little-endian number where each position represents a digit between 0 and $2^{64} - 1$. The carry stored in `high` is added to the following 64 bit limb of `t`. The modulo trick is performed by multiplying the overflowing $h1$ limb with `r54`, the pre calculated value $r54 = r1 >> 2*5$. The same mechanism is used for $h2 * r54$:

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

4 Code verification/audition

The final phase of this report will perform a formal verification of the assumptions made during the implementation. This verification will take the form of mathematical formulas that guarantee that the assumptions regarding bit spanning and lower and upper bounds are met.

5 Conclusion

References

[1] José Bacelar Almeida et al. "Jasmin: High-Assurance and High-Speed Cryptography". In: Oct. 2017, pp. 1807–1823. DOI: 10.1145/3133956.3134078.

[2] Sean Gulley et al. *Intel ® SHA Extensions New Instructions Supporting the Secure Hash Algorithm on Intel ® Architecture Processors*. URL: https://software.intel.com/sites/default/files/artic 2013.