Universidade do Minho

Mestrado em Engenharia Informática

Tecnologia Criptográfica

Auditing a Poly1305 MAC implementation in Jasmin

Miguel Miranda Quaresma
A77049

January 20, 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 [3], such as Intel's instruction for SHA-256: `sha256rnds2`. The development of performance focused cryptoprimitives such as the Poly1305 MAC used in conjunction with frameworks such as Jasmin [1] has represented an attempt to reverse this tendency.

2 Poly1305 explained

Poly1305 is a message authentication code(MAC) that guarantees the integrity and authenticity of messages by generating a 16 byte digest using a mechanism similar to universal hash functions. Poly1305 can also be paired with a symmetric cipher([2], [6]) that guarantees the confidentiality of the information exchanged. As stated Poly1305 works similarly to a universal hash function, as such it 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 used a 256 bit key that can be derived either via a Password-Based Key Derivation Function(PBKDF) or by combining a long-term key with a nonce used as input to the same symmetric cipher used to encrypt the message. The key is then split up into two blocks of 128 bits each, the first block being used for the parameter r 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, using this blocks as coefficients of the of the aforementioned polynomial, applying the following algorithm:

```
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 to prevent this it is useful to resort to modular arithmetic. With performance being a main focus of Poly1305 it is important that the modular arithmetic is performed as efficiently as possible one. To allow for this to happen the prime chosen must be in the form $2^n - p$, with p a small number, in the case of Poly1305 the prime $2^{130} - 5$ was chosen which makes it possible to perform schoolbook multiplication with delayed carry propagation as we will see.

2.1 (Schoolbook) Multiplication

Following the previously described algorithm, after adding each message block to the current h value, h is multiplied with r. The operation is performed by dividing r and h in five 4 byte blocks, called limbs, which are seen as single digits:

	(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**

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 then achieved by multiplying the (overflowing) limbs by 5, which is congruent with 2^{130}:

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

We can now see why choosing a prime in the form of $2^n - q$ is paramount for fast modular arithmetic: it allows for modular reduction to be performed by multiplying by q(=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 however, a caveat in this process: since h and r limbs can span 32 bits the resulting multiplication can take up 64 bits preventing the delay of carry propagation and hindering performance significantly as a consequence. To prevent this from happening Poly1305 removes 22 bits from r:

```
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 a h limbs spans 32 bits, h*r will only take up 60 bits (28+32)..

After processing all the message blocks the MAC is generated by adding k and performing a full modular reduction:

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

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 [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)
```

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

3.1 Setup

The parameters used by 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 sets h to 0, loads r and pre computes r54 = r*5/4. This pre computation is important as it allows for the value to be reused without needing to calculate it everytime.

```
    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;
```

3.2 MAC Calculation

The message digest (MAC) is calculated by calling `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 2:

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

Each 16 byte block is loaded from the memory location pointed to by `in` and added to the accumulator `h` by calling:

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

which appending a 1 byte to each message block previously. The multiplication and modular reduction are performed by calling:

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

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

3.2.1 Modular multiplication

To understand the mechanism used by `mulmod` to perform modular multiplication, it's important to note that this implementation 64 bit limbs `(reg u64[3] h, reg u64[2] r, reg u64 r54)` instead of the 32 bit ones described earlier, 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 considered we can now examine the way `mulmod` works. 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 $\mathtt{h} = (h * r) \mod 2^{130}$, 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, serving as a more trustworthy methodology to traditional software testing. There are many tools available to perform formal verification of cryptographic and other software systems(see [4] and [5]) 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.

4.1 First assumption: h[0] and r[0] upper bounder

```
//   r[0] upper bound is 0x0ffffffc0fffffff (60 bits)
//   h[0] upper bound is 2**64-1 (64 bits, all bits can be set)
//   the resulting multiplication requires 124 bits, 60 + 64:
//   - 0x0ffffffc_0ffffffe_f0000003_f0000001
```

The most important assumption that serves as the basis for every other is the upper bound for each `r` and `h` limb. This bounds are represented as hexadecimal values since they allow for a more comprehensive verification by performing the arithmetic without having to convert between representations. To understand upper bound for `r[0]` we must remember that Poly1305 clears 22 bits from `r` previous to using it's value. It does so by applying the following mask `0x0ffffffc0fffffff` with the logic operator &(and) to the upper 64 bit limb:

```
r[0] &= 0x0ffffffc0fffffff;
```

As such, due to the very nature of the &(and) operation, `r[0]` will have at most the same number of 1 bits as the mask applied to it. In this case the top 4 bits($0x0 = 0000_b$) which makes it so that `r[0]` can span, at most, 60(64-4) bits, which confirms the assumption.

The upper bound for `h[0]` corrresponds to the case where the limb has all its bits set to 1. Being 64 bits in size means they can have all 64 bits set which sets the upper bound at $(2^{64} - 1)_{10}$.

A multiplication by a power of 2 can be performed by shifting values to the left as many positions as the exponent. Taking $2^{64} - 1$ and multiplying it by `r[0]` value would then correspond to shifting `r` to left by 64 bits, requiring $64 + 60 = 124$ bits, and subtracting `r[0]` to the result: $r[0] << 64 - r[0]$ which verifies the assumption about the multiplication of `h[0]` and `r[0]`.

4.2 Second assumption: h[0] and r54 upper bound

```
//   r[1] * 5/4 is precomputed (in clamp function) and it is in r54 variable
//   r[1] upper bound is : 0x0ffffffc0ffffffc
//
//   we compute r54 in the following manner
//     - r54 = r[1] + r[1]/4 where /4 is performed by a shift to the right
//       by 2 (first 2 bits of r[1] are 0 (0x..fc))
//     - r54 upper bound is 0x13fffffb13fffffb which requires 61 bits
//
//   the resulting multiplication of (h[1] * r54) requires 125 bits, 61 + 64:
//     - 0x13fffffb_13fffffa_ec000004_ec000005
//
//   since we are adding this partial result to the previous one we get
//   (124 bits + 125 bits):
//     - 0x0ffffffc_0ffffffe_f0000003_f0000001 +
//     - 0x13fffffb_13fffffa_ec000004_ec000005 =
//     - 0x23fffff7_23fffff9_dc000008_dc000006 which requires 126 bits
//       (atm there are only 2 bits left in t[1])
```

Let's take a look at the upper bound for `r54`. The first thing to remember is that `r54=r[1]*5/4` so we can derive it's upper bound from `r[1]`. `r[1]`'s upper bound is similar to that of `r[0]` with the small variation that, additionally to the 10 bits removed from `r[0]`, `r[1]` also has it's bottom 4 bits removed:

```
r[1] &= 0x0ffffffc0ffffffc;
```

However, `r[0]` still requires, at least, 60(=64-4) bits to be represented. `r54` is a result of multiplying `r[1]` by 5/4 which is equivalent to:

$$r54 = r[1]/4 + r[1] = r[1](1 + 1/4) = r[1] * 5/4$$

The division by 4 is performed by shifting `r[1]` two bits to the right, since it already has it's 4 bottom bits cleared this operation doesn't change the upper limit for the bit count however, adding r[1] may involve carry which, in the limit, will set the value of `r54` to 61(= 60 + 1) bits.

	(r[1]>>2)	0x3ffffff03ffffff
+	(r[1])	0x0ffffffc0ffffffc
	(r54)	0x13fffffb13fffffb

Multiplying `h[1]` by `r54`, when `h[1]` is in the upper bound($2^{64}-1$) is equivalent to shifing `r54` 64 bits to the left and subtracting `r54`, which is the same as performing an addition with the two's complement of `r54`(=-`r54`):

		(r54¡¡64)	`0x13fffffb13fffffb0000000000`
`0xec000004ec000005.`	+	(-r54)	`0xec000004ec000005`
		(r54)	`0x13fffffb13fffffbec000004ec000005`

Observing the code it's visible that this value(`high,low=r54*h[1]`), as explained before, will be added to the accumulator `t`, in this case at index 0, to the previous value which required, at most, 124bits. This means that the addition will take up, at most, 126 bits:

		(`low`)	`0x13fffffb13fffffaec000004ec000005`
	+	(`t[0]`)	`0x0fffffffc0fffffffef0000003f0000001`
		(`t[0]`)	`0x`

5 Conclusion

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

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] Daniel J. Bernstein. "The Poly1305-AES Message-Authentication Code". In: *Fast Software Encryption*. Ed. by Henri Gilbert and Helena Handschuh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 32–49. ISBN: 978-3-540-31669-5.

[3] 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/article/402097/intel-sha-extensions-white-paper.pdf. 2013.

[4] Inria. *The Coq Proof Assistant*. URL: https://coq.inria.fr.

[5] IMDEA Software Institute and Inria. *EasyCrypt: Computer-Aided Cryptographic Proof*. URL: https://www.easycrypt.info/trac.

[6] F. De Santis, A. Schauer, and G. Sigl. "ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. Mar. 2017, pp. 692–697. DOI: 10.23919/DATE.2017.7927078.

0x0fffffc0ffffff¡¡64 - 0x0fffffc0ffffff = 0xffffffc0fffffff0000000000000000 - 0x0fffffc0ffffff = 0xffffffc0fffffef0000003f0000001