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 message authenticator that generates a message authentication code for a given input and secret key pair 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 verification 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 MACs 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] and formally verified implementations, has represented an effort with potential to reverse this tendency.

2 Poly1305 explained

Poly1305 is a message-authentication code that generates a 16 byte digest for a variable length message by evaluating a polynomial over a prime field, similar to the way universal hash functions work. It can also be paired with a symmetric cipher([2], [6]) that guarantees the confidentiality of the information exchanged. As stated, Poly1305 is a polynomial-evaluation MAC that works over the prime field $\mathbf{GF(}2^{130}-5\mathbf{)}$, thus the name (Poly**1305**). The general formula for Poly1305 can be expressed by the following expression:

$$Poly1305(m,r,k) = (m1 * r^4 + m2 * r^3 + m3 * r^2 + m4 * r + k) \mod 2^{130} - 5$$

where mi is the i-th block of the message m. Being an authenticator, a secret is required, in this case a 256 bit key is used, which can be computed 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. To allow for variable length messages these are divided in 16 byte blocks, appending each block with a 1 byte(00000001) to prevent forgery, and using this blocks as coefficients of the aforementioned polynomial:

```
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

2.1 (Schoolbook) Multiplication

In Poly1305 each message block is added to an accumulator h and multiplied with r. This product involves dividing r and h in five 4 byte blocks, called limbs, which are seen as single digits and are multiplied to get the value of h at each iteration:

	(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**

As stated, the operations are performed modulo $2^{130} - 5$ which means that values overflowing 130 bits must be reduced. Since the modular reduction is performed with 32 bit limbs and the closest multiple to 130 is 128, before being reduced the limbs are shifted 2 bits to the right. The reduction is achieved by multiplying the limb by 5, which is congruent with 2^{130} in the field $\mathbf{GF}(2^{130} - 5)$:

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

It is now obvious that choosing the right prime is paramount for performance in modular arithmetic: the reductions are just a matter of multiplying each value by a small number such as 5, which can be done extremely fast and reduces the chances of overflowing the registers (**n.b.** this isn't a full modular reduction by $2^{130} - 5$ which will 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 4 byte blocks called limbs prior to calculating their product, and carry propagation is delayed until it is mandatory. 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 the Poly1305 specification removes 22 bits from r by applying a mask to each limb:

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

The resulting r limbs will have, at most, 28 bits set (32-4). As a consequence, even if h limbs span 32 bits, $h * r$ will only take up 60 bits (28+32). After processing all the message blocks the MAC is generated by adding the rest of 256 bit key, k, and performing a full modular reduction:

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

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

3 Jasmin Poly1305 Implementation

The performance focus of Poly1305 allows for implementations that take advantage of modern hardware characteristics, with high parallelizability that can surpass, performance-wise, hash based MACs. It's important however for these implementations not to disregard the assurance needs of cryptographic software. For this reason we will examine the use of the Jasmin framework, which has performance and assurance as the main focus, for implementing Poly1305. Jasmin is based on qhasm, however it uses Coq proof assistant to formally verify the (assembly) code generated by the (Jasmin) compiler, providing high assurance high performance cryptograhic code [1]. In this case the function that represents the entry point for code developed 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 memory pointers to the 16 byte output code, 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 initial values for this parameters are set by calling:

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

This function sets h to 0, loads r and pre computes r54 = r*5/4. This pre computation removes the need to calculate the value at every use.

```
    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 input in 16 byte blocks, adding each block to `h` and multiplying the result with `r` $\mod 2^{130}$, 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]
```

It's important to note that, to prevent forgery, `add_bit` also appends a 1 byte to each message block. The modular multiplication is 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

The mechanism used by `mulmod` to perform modular multiplication is a slight variation of the one showcased earlier. This is is because the limbs, in this case, are 64 bits in size`(reg u64[3] h, reg u64[2] r, reg u64 r54)` as opposed to 32 bits ones, hence the schoolbook multiplication takes the form of:

$$\begin{array}{cccc}
 & (2^{128}) & (2^{64}) & (2^{0}) \\
 & \text{h2} & \text{h1} & \text{h0} \\
* & & \text{r1} & \text{r0} \\
\hline
 & \text{h2*r0} & \text{h0*r1} & \text{h0*r0} \\
+ & & \text{h1*r0} & \textbf{h1*5*r1} \\
+ & & \textbf{h2*5*r1} &
\end{array}$$

which is equivalent to:

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

To perform the multiplication, two variables are declared in `mulmod`: `low` which is used to store the `h` limbs at each 64 bit multiplication, since the h values are used multiple times and can't be overwritten(this will not stand for the $h2 * r0$ product as we will see); `high` which is used to hold the carry of the products between each `r` and `h` limb. Because the code for the first two 64 bit blocks is similar, this analysis will only contemplate the portion corresponding to $h0 * r0 + h1 * r54$, as the other one ($2^{64} * (h0 * r1 + h1 * r0 + h2 * r54)$) is performed by applying the same general principles. The following code snippet performs the $h0 * r0$ operation:

```
low = h[0];
high, low = low * r[0];
t[0] = low;
t[1] = high;
```

The result of the multiplications are (temporarily) stored in the `t` array, where each position represents a digit between 0 and $2^{64} - 1$:

$$\begin{array}{ccc}
2^{128} & 2^{64} & 2^{0} \\
\texttt{t[2]} & \texttt{t[1]} & \texttt{t[0]}
\end{array}$$

In this case, since `high` contains the carry from the `low*r[0]` product, it is assigned to `t[1]`, as it overflows the first 64 bits. As is the case for subsequent `high` values. will follow the same guideline. The $h1 * r1$ operation is similar to the previous one however, in this case, since the value overflows 130 bits it must be reduced by multiplying the result by 5 and dividing it by 4. This value is pre computed by the `clamp` function and stored in `r54`, resulting in the following code:

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

After performing all the smaller (order) operations, $h2 * r0$ is calculated (inplace): `h[2] *= r[0];` and the value of `h` is merged with that of `t`. Considering that `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. To do so requires that 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

Before adding k, the last 128 bits of the key, a final reduction $\mod 2^{130}$ is performed by calling:

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

To perform the reduction, h is assigned to g which is reduced $\mod 2^{130} - 5$ by adding 5 and propagating the carry:

```
g = h;

cf, g[0] += 5;
cf, g[1] += 0 + cf;
_, g[2] += 0 + cf;
```

After this a check is performed to verify whether g overflows 130 bits, which occurs when the value of `g[2]` has more than two bits sets after carry propagation. To perform this check a mask corresponding to the two's complement of `g[2]`>>2 is applied to the two lower limbs:

```
  mask = -g[2];

  g[0] ^= h[0];
  g[1] ^= h[1];

  g[0] &= mask;
  g[1] &= mask;
```

This mask will be 0 when g[2] has less than two bits set and $2^{64} - 1$ otherwise. The reduction is then peformed by combining the two lower limbs of h and g via the xor operator and storing the result in h:

```
h[0] ^= g[0];
h[1] ^= g[1];

// if bit == 1
//h[0] = h[0] ^ (g[0] ^ h[0]) = h[0] ^ h[0] ^ g[0] = 0 ^ g[0] = g[0];
//h[0] = h[1] ^ (g[1] ^ h[1]) = h[1] ^ h[1] ^ g[1] = 0 ^ g[1] = g[1];
// else
//h[0] = h[0] ^ (g[0] ^ h[0]) & 0 = h[0] ^ 0 = h[0];
//h[1] = h[1] ^ (g[1] ^ h[1]) & 0 = h[1] ^ 0 = h[1];
```

The last 128 bits of the original key value are added to the value returned by freeze and the digest is stored in the memory location pointed to by out:

```
h = freeze(h);
s = load(k);
h = add_bit(h, s, 0);
store(out, h);
```

4 Code verification/audition

Formal verification is one of the most important parts of developing software systems used in critical environments, as it provides mathematical proof of the correctness of an implementation without needing to test all the possible inputs, serving as a more trustworthy methodology to traditional software testing. There are many tools available to perform formal verification of cryptographic software and other systems(see [4] and [5]) however, the proof hereby presented is in the form of mathematical calculations that should serve only as a possible template from which to build formal proofs with such tools. The assumptions made in the previous implementation are present in the form of comments, mainly on the mulmod function, as pointed out by the developer:

```
// 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]
```

To keep this verification short and concise only the verification of the first two assumptions will be presented.

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 first assumption made plays an integral role in every other that follows and it relates to the upper bounds for each `r` and `h` limb. This bounds are represented as hexadecimal values which allows for modular arithmetic to performed without having to convert between different numerical representations. The upper bound for `r[0]` is set taking into account that Poly1305 clears 22 bits from `r`, previous to using it's value. It does so by applying the mask `0x0ffffffc0fffffff` via the logic operator &(and) to the upper 64 bit limb:

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

As such, due to the very nature of the &(and) operator, `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($0x0=0000_b$) of `r[0]` are cleared so it can span, at most, 60(64-4) bits.

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 that when all the bits are set the value of `h[0]` is $(2^{64}-1)_{10}$, which confirms the upper bound set for this limb.

Multiplying `r[0]` by `h[0]`, when `h[0]` is equal to $(2^{64}-1)_{10}$, is the same as: `r[0]*`2^{64}`-r[0]`. This value can be calculated by shifting `r[0]` 64 bits to the left and subtracting `r[0]` to the result, requiring a total of 60+64=124 bits:

```
       (r[0]<<64)   0x0ffffffc0ffffffe0000000000000000
  +      (-r[0])    0x0000000000000000f0000003f0000001
  ------------------------------------------------------
      (t[1],t[0])   0x0ffffffc0ffffffef0000003f0000001
```

confirming the assumption made.

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 0x13ffffffb13ffffffb 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])
```

The next assumption sets 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]`. The mask applied to `r[1]` removes an additional 10 bits in relation to `r[0]` however, the upper bounds for both values are similar since `r[1]` still requires 60 bits to be represented. Using this upper bound we can now proceed to determine `r54`'s upper bound. `r54` is the result of multiplying `r[1]` by 5/4, which is equivalent to:

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

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 amount of 1 bits `r54` can have set. However, adding `r[1]` may involve carry will make it so that the upper bound for `r54` takes requires $61(= 60 + 1)$ bits to be represented.

$$\begin{array}{rcl} & \texttt{(r[1]>>2)} & \texttt{0x03ffffff03ffffff} \\ + & \texttt{(r[1])} & \texttt{0x0ffffffc0ffffffc} \\ \hline & \texttt{(r54)} & \texttt{0x13fffffb13fffffb} \end{array}$$

The upper bound for `h[1]` is the same as for `h[0]` thus, multiplying `h[1]` by `r54`, when `h[1]` equals $2^{64} - 1$, is equivalent to shifting `r54` 64 bits to the left and subtracting `r54`, which is the same as performing an addition with the two's complement of `r54`: `0xec000004ec000005`.

$$\begin{array}{rcl} & \texttt{(r54<<64)} & \texttt{0x13fffffb13fffffa0000000000000000} \\ + & \texttt{(-r54)} & \texttt{0x0000000000000000ec000004ec000005} \\ \hline & \texttt{(r54)} & \texttt{0x13fffffb13fffffaec000004ec000005} \end{array}$$

As in the previous operation, the value of the 64 least significant bits of `r54*h[1]` are stored in `low` and the carry is stored in `high` This values are added to `t` and the result takes up 126 bits:

$$\begin{array}{rcl} & \texttt{(high,low)} & \texttt{0x13fffffb13fffffaec000004ec000005} \\ + & \texttt{(t[1],t[0])} & \texttt{0x0ffffffc0fffffffef0000003f0000001} \\ \hline & \texttt{(t[1],t[0])} & \texttt{0x23fffff723fffff9dc000008dc000006} \end{array}$$

verifying the assumption that was made about the value of `t[0]` and `t[1]`.

4.3 Final assumption

The verification of the remaining assumptions follows the same guidelines however, the upper bound for `h[2]` varies between the first and remaining iterations, being 0x3 and 0x1 respectively.

5 Conclusion

Poly1305 is a message-authentication code developed with a focus on performance. Consequently, every aspect of it's specification has this objective in mind, from the choice of performing the polynomial evaluation over a prime field instead of a binary field to the use of a prime that allows for modular arithmetic to be performed in the most efficient manner. Additionaly, the use of Poly1305 in conjunction with AES, besides being parallelizable, offers a security level close to that of AES, meaning that an attacker can only obtain a sucessfull forgery if it can break AES. Thus, to guarantee this security and performance it's paramount to implement Poly1305 using a high assurance framework such as Jasmin, using formal verification as a means to prove the stated properties. Therefore, the work hereby presented, accomplished the goal of providing a framework from which a formal verification for the analyzed implementation can be constructed.

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`.