SOCIAL

atom feed
twitter
github

CATEGORIES

Android
Challenge
Cryptography
Development
Exploitation
Fuzzing
Life at Quarkslab
Maths
PenTest
Program Analysis
Programming
ReverseEngineering
Software

TAGS

Reversing a Finite Field Multiplication Optimization

Date Wed 20 April 2016 By Jean-Baptiste Bédrune Marion Videau Category ReverseEngineering. Tags cryptography mode aes-gcm optimization arithmetic

An optimization for the finite field multiplication on 128-bit elements for AES-GCM exists whose explanation was not published, preventing any further application with different parameters. We reverse engineered the result to 1) get the explanation and 2) be able to apply it with other parameters.

Introduction

The goal of this article is to explain what this piece of code does.

```
vmovdqa    T3, [W]
vpclmulqdq T2, T3, T7, 0x01
vpshufd    T4, T7, 78
vpxor      T4, T4, T2
vpclmulqdq T2, T3, T4, 0x01
vpshufd    T4, T4, 78
vpxor      T4, T4, T2
vpxor      T1, T1, T4  ; result in T1
```

In fact, we are not completely in the dark as we know it performs, in an optimized way, a finite field multiplication used in AES-GCM. We will show how we reconstructed the higher level algorithm behind the optimization. The reversing allowed us to show the equivalence between seemingly different pieces of codes, understand what the constants represented and eventually apply the optimization with different parameters.

Context

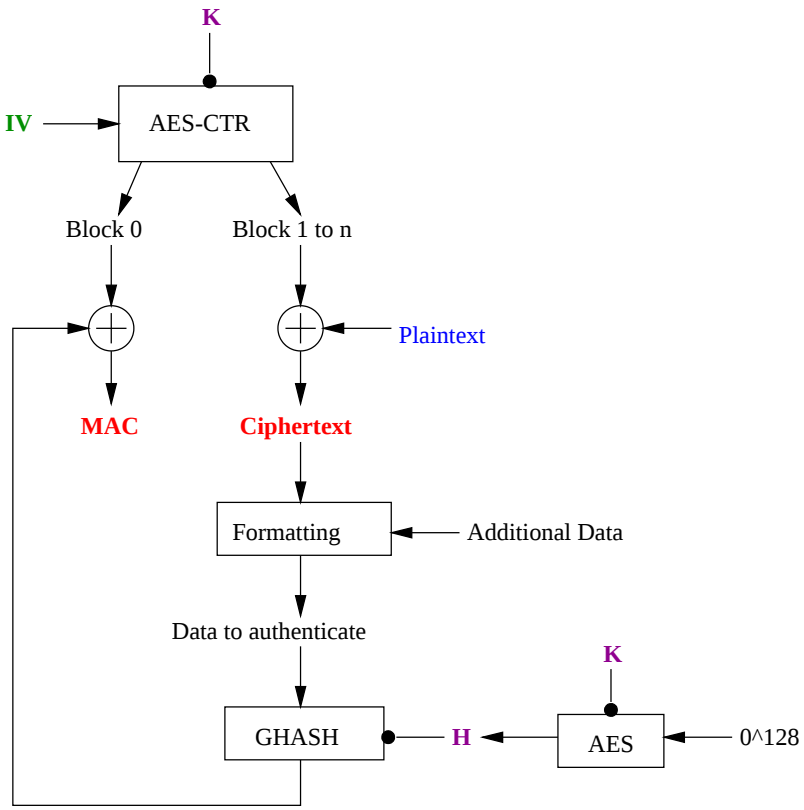
The Galois/Counter Mode of operation (or GCM) has been standardized by NIST in 2007 [GCM2007] as an authenticated encryption mode to be used with AES [AES2001]. Its performance is notably good when used in *software* on usual desktops since recent Intel architectures propose the AES-NI [1] and the CLMUL instruction sets [2].

To avoid having to dive into small details of the standard, we will simply sum up the main characteristics of the AES-GCM mode. The general principle behind the construction of the mode is the so-called Wegman-Carter construction for a MAC (Message Authentication Code) —a MAC being the symmetric cryptographic algorithm that provides authentication of the message origin (also known as cryptographic integrity) between two parties. To build such a MAC you need two ingredients:

- A *universal family of hash functions* [3], i.e. a set of keyed non-cryptographic hash functions having a low probability of collision when the key is chosen randomly. In AES-GCM, this universal family is called GHASH (the Galois part of the name) and its key is H given by the 128-bit all-zero bloc encrypted with AES under a key K. The data to

authenticate (the ciphertext plus some additional data) is hashed with GHASH which outputs a 128-bit block.

- A keystream to XOR with the output of GHASH. In AES-GCM, the keystream is obtained by AES in counter mode (the Counter part of the name) keyed with the key K. The first block of keystream is reserved to encrypt the output of GHASH, the rest of the keystream is used to encrypt the plaintext, giving the ciphertext—which is the input to GHASH, if you followed up to now; in case you did not, a little graphic help thereafter.



The very heart of GHASH relies on multiplying (in a finite field sense) the 128-bit blocks of data to authenticate with powers (in a finite field sense) of H. A finite field being also called a Galois field, you know why Galois—Évariste Galois (1811-1832)—has been invoked.

The input being m 128-bit blocks $X_1, \ldots, X_m$, the universal hashing is performed under the key H in the following manner:

$$\mathrm{GHASH}(X_1, \ldots, X_m) = X_1 \bullet H^m \oplus X_2 \bullet H^{m-1} \oplus \cdots \oplus X_m \bullet H,$$

where $\bullet$ stands for the finite field multiplication and $\oplus$ is the finite field addition.

How Data Are Represented

As it is particularly crucial not to misunderstand the slightest bit of any constant in representing a finite field, we need to take a little time to explain how data are represented. Furthermore, in the case of GCM, the representation is not even *usual*.

Binary Finite Field elements into Binary Strings

We work in the binary finite field GF(2^{128}). Elements are represented as binary polynomials of degree at most 127. Such elements are stored through the sequence of their binary coefficients, which means an element is represented as a 128-bit word, where the coefficient of the degree i monomial is stored at position i. Then, the most significant bit represents the coefficient of the monomial of degree 127 and the least significant bit represents the constant of the polynomial.

Example: The bit string $a = a_{127}a_{126} \dots a_0$ stores the sequence of coefficients of the polynomial $a(X) = a_{127}X^{127} + a_{126}X^{126} + \cdots + a_0$. For instance, we consider a finite field element P_1 which is represented by the polynomial $P_1(X) = X^7 + X^2 + X + 1$. The polynomial—therefore the finite field element P_1, is stored on 128 bits as the bit string `(0^120)10000111`, where `(0^120)` stands for `0` repeated 120 times.

The **finite field addition** of two elements is simply carried out by the binary sum of the coefficients of the monomials having the same degree. It corresponds to the bitwise XOR of the two 128-bit words representing the polynomials. It means, we denote by $+$ the addition of two polynomials and we denote by $\oplus$ the addition of two elements of the finite field, as it is a bitwise XOR on the bit string representation.

Example: The finite field addition of $P_1(X) = X^7 + X^2 + X + 1$ and $P_2(X) = X^6 + X^5 + X + 1$ is represented by `(0^120)10000111` $\oplus$ `(0^120)01100011 = (0^120)11100100`.

As we work with binary polynomials, subtraction is the same as addition and in the following we will not use any minus signs.

The **polynomial multiplication** of two polynomials is the usual one you already know.

Example: The polynomial multiplication of $P_1(X) = X^7 + X^2 + X + 1$ and X^2 equals $X^9 + X^4 + X^3 + X^2$. This example shows that the polynomial multiplication by X^n corresponds to a left shift by n positions. If the degree of the result is greater than the size of the word you use to represent it, then you lose coefficients in the void.

The **finite field multiplication** of two elements, that we denote by $\bullet$, corresponds to the modular multiplication of the two corresponding polynomials modulo the defining polynomial of the finite field. The GCM standard specifies that the defining polynomial one should use is $P(X) = X^{128} + R(X)$ with $R(X) = X^7 + X^2 + X + 1$.

Example: The finite field multiplication of $P_1(X) = X^7 + X^2 + X + 1$ and $P_3(X) = X^{121}$ equals $P_1(X) \times P_3(X) \mod P(X)$, which equals $X^{128} + X^{123} + X^{122} + X^{121} \mod P(X)$. This means $(P_1 \bullet P_3)(X) = X^{123} + X^{122} + X^{121} + X^7 + X^2 + X + 1$, hence `(0^120)10000111` $\bullet$ `0000001(0^121) = 0000111(0^113)10000111`.

The **finite field multiplication by X^n** is an important special case. It simply corresponds to the left shift by n positions, when the degree of the polynomial to multiply is smaller than $127 - n$ (when this degree is bigger than $128 - n$ it resorts to the former general case finite field

multiplication).

Example: Multiplying $P_1(X) = X^7 + X^2 + X + 1$ with $P_4(X) = X^2$ equals $P_1(X) \times P_4(X) \bmod P(X)$, which equals $X^9 + X^4 + X^3 + X^2$. Hence `(0^120)10000111 • (0^125)100 = (0^118)1000011100`.

The **polynomial division** is denoted by div and is the usual Euclidean division (like the integer division but for polynomials).

Example: Dividing $P_6(X) = X^9 + X^4 + X^3 + X^2 + X + 1$ by X^2 in the polynomials gives: $P_6(X) \mathrm{\ div\ } P_4(X) = X^7 + X^2 + X + 1$. This example shows that the polynomial division by X^n corresponds to a right shift by n positions. Of course, you lose all the coefficients of indices below zero in the void.

The **finite field division** of a by b, that we denote by a/b, corresponds to $a \bullet b^{-1}$ where b^{-1} is computed thanks to the extended Euclidean algorithm. As we represent elements by polynomials, the operation is denoted by $a(X)/b(X) \pmod{P(X)}$, which means we compute first the modular inverse of $b(X)$ modulo $P(X)$ and then we multiply it modulo $P(X)$ by $a(X)$.

Example: The finite field division of $P_6(X) = X^9 + X^4 + X^3 + X^2 + X + 1$ by $P_4(X) = X^2$ is represented by $P_6(X)/P_4(X) \pmod{P(X)}$. The extended Euclidean algorithm provides the value of $P_4^{-1}(X) \pmod{P(X)} = X^{127} + X^{126} + X^6 + X^5 + X$. Then $(P_6/P_4)(X) \pmod{P(X)}$ equals $(X^9 + X^4 + X^3 + X^2 + X + 1) \times (X^{127} + X^{126} + X^6 + X^5 + X) \pmod{P(X)}$. The result is $X^7 + X^4 + X^3 + X^2 + X + 1$. Hence `(0^118)1000011111 / (0^125)100 = (0^120)10011111`.

The **finite field division by** X^n is the counterpart of the multiplication case. It corresponds to the right shift by n positions, when the degree of the smallest degree monomial of the polynomial to divide is bigger than n (when this degree is smaller than $n - 1$ it resorts to the general case division).

Example: Dividing $P_5(X) = X^9 + X^4 + X^3 + X^2$ by $P_4(X) = X^2$ equals $P_5(X)/P_4(X) \bmod P(X) = X^7 + X^2 + X + 1$. Hence `(0^118)1000011100} / (0^125)100 = (0^120)10000111`.

Beware of notions and pieces of notations, especially for division. We denote by div the polynomial division and by $/$ the finite field division. The only case where $a(X) \mathrm{\ div\ } b(X)$ corresponds to $a(X)/b(X) \pmod{P(X)}$ is the case where $a(X)$ is a multiple of $b(X)$ of degree less than 127. Otherwise, we deal with two distinct notions and computations.

Binary Strings into Binary Finite Field Elements

Suppose you are given a binary file and you want to apply AES-GCM to it. How would you *embed* the bit strings into elements of a finite field? The straightforward way is to consider the bit string as the sequence of the coefficients of the polynomial representing an element of the finite field we are working in. This leaves to the designer the choice of the bit numbering to use, e.g. either considering the bit string $a = \mathtt{11001001}$ to be $a = a_7a_6a_5a_4a_3a_2a_1a_0$ (most significant bit is leftmost) or $a = a_0a_1a_2a_3a_4a_5a_6a_7$ (most significant bit is rightmost).

The most usual representation used when manipulating elements in a finite field is the one where most significant bit means leftmost bit and least significant bit means rightmost bit. We used it beforehand to illustrate the representation of finite field elements. The GCM standard counterintuitively specifies that bit strings have to be considered with the opposite convention. It means that with GCM, a 128-bit block has to be numbered $a = a_0a_1a_2a_3 \dots a_{127}$. This implies that on most cases when the two conventions have to be used simultaneously, input bit strings have to be *reflected* first before being applied finite field operations and the result be reflected back, to comply with the standard (one can find an analysis of such a choice by Rogaway in [Rog2011], Remark 12.4.4, p.130). It is the case with all of the Intel's implementations.

Available Instructions for Binary Finite Field Arithmetic

As we have seen, adding two finite field elements is easy, we use the bitwise XOR. Some cases of multiplications or divisions are easy, we use left or right shifts. Until recently, the general case multiplication or division resorted to special arithmetic operations to be implemented in software. The situation ended for the multiplication with the advent of the `pclmulqdq` instruction which computes the polynomial multiplication.

Efficiency Evolution of the PCLMULQDQ Instruction

The multiplication of two binary polynomials can be efficiently carried out with the `pclmulqdq` instruction which was made available by Intel since the Westmere processor line in 2010, explicitly to improve the efficiency of AES-GCM, coming along the AES-NI instruction set. CLMUL stands for *carry-less multiplication* as in polynomial arithmetic, contrary to the usual integer arithmetic, there is no carry to take care of. The instruction typically takes as inputs two 64-bit words and outputs a 128-bit result.

Quite ironically, its carry-less characteristic did not make the `pclmulqdq` instruction very efficient at first, compared to a standard integer multiplication. If we refer to Fog's Instruction tables [Fog2016], the latency and reciprocal throughput of `pclmulqdq` have evolved significantly since 2010.

Year (approx.)	CPU Name	Latency	Reciprocal throughput
2010	Intel Westmere	12	8
2011	Intel Sandy Bridge	14	8
2012	Intel Ivy Bridge	14	8
2013	Intel Haswell	7	2
2014	Intel Broadwell	5	1
2015	Intel Skylake	7	1
2011	AMD Bulldozer	12	7
2012	AMD Piledriver	12	7
2014	AMD Steamroller	11	7
2013	AMD Jaguar	3	1

This evolution explains why computation involving `pclmulqdq` are subjected to various optimization techniques depending on the targeted platform. For instance a multiplication of two 128-bit words into a 256-bit word requires either 4 `pclmulqdq` instructions with a classical approach or 3 `pclmulqdq` instructions with a Karatsuba algorithm. The latter was preferred

with a slow `pclmulqdq` instruction and a classical approach preferred with a faster `pclmulqdq`.

For the same reason, various optimization strategies were considered for the modular reduction. Why optimize the modular reduction? In fact, once the multiplication of two polynomials is carried out, the remainder is obtained through the division (the Euclidean one) which outputs a quotient and a remainder. As division is usually more expensive than multiplication several strategies were devised to trade divisions for multiplications, which are especially efficient when one works entirely with the same modulus as it is the case with finite field computations.

A Former Optimization of the Modular Reduction

In 2009, in [GK2010], Gueron and Kounavis proposed an optimization of the reduction modulo $P(X)$ which only uses shifts and XORs and was taking place after the reflection of the inputs and their multiplication.

Their technique relied on two steps. First they applied a reduction algorithm known as the Barret's algorithm (see [BZ2010] 2.4.1) which involves 2 carry-less multiplications. The algorithm, adapted to the case of binary polynomials of degree less than 256 to be reduced by a polynomial of degree 128, is given below.

Require:

- X^{128}, the Barrett's basis
- U, the input polynomial to reduce, of degree at most 255
- P, the reduction polynomial of degree 128
- $P' = X^{256} \operatorname{div} P$

Ensure: $T = U \bmod P$

1: **function** BarrettReduction(U, X^{128}, P, P')
2: $\quad Q \leftarrow \left((U \operatorname{div} X^{128}) \times P'\right) \operatorname{div} X^{128}$
3: $\quad T \leftarrow U + Q \times P$
4: $\quad$ **return** T
5: **end function**

Then they took advantage of the special structure of the reduction polynomial to devise an optimization only relying on shifts and XORs. Indeed, going into details, for $P(X) = X^{128} + X^7 + X^2 + X + 1$, we have $P' = X^{256} \operatorname{div} P = P$. Then Line 2 of the function (computation of the quotient Q) can be rewritten as:

```
Input: 256-bit operand [X3 : X2 : X1 : X0]
A = X3 >> 63
B = X3 >> 62
C = X3 >> 57
D = X2 ⊕ A ⊕ B ⊕ C
Output: D
```

Line 3 of the algorithm is in fact equivalent to $T = (U \bmod X^{128}) + (Q \times (P + X^{128})$ as the remainder has degree less than the modulus. This means we only consider the 128 least

significant coefficients of U and we multiply Q by $X^7 + X^2 + X + 1$. This gives the following optimization:

```
Input: 256-bit operand [X3 : X2 : X1 : X0] and D computed beforehand
[E1 : E0] = [X3 : D] << 1
[F1 : F0] = [X3 : D] << 2
[G1 : G0] = [X3 : D] << 7
Output: [X3 ⊕ E1 ⊕ F1 ⊕ G1 ⊕ X1 : D ⊕ E0 ⊕ F0 ⊕ G0 ⊕ X0]
```

This optimization works on already reflected inputs and made sense when the `pclmulqdq` instruction was rather slow as the aim was globally trying to avoid too much of it. The trend of a faster `pclmulqdq` made it interesting to consider another optimization.

An optimization that Needs Explanations

In 2012, in [OG2012] and as we will see equivalently in [Gue2013], Intel proposed a new optimization of the modular reduction. It involved working directly on straight inputs (i.e. non reflected) and the call to `pclmulqdq`.

The optimization described by Ozturk and Gopal in OG2012 is called *PCLMULQDQ-based reduction* and is reproduced thereafter. Very few explanations are available in the document and the value of POLY2 is not given.

```
;first phase of the reduction
movdqa %%T3, [POLY2 wrt rip]
movdqa %%T2, %%T3
pclmulqdq %%T2, %%GH, 0x01
pslldq %%T2, 8

pxor %%GH, %%T2

;second phase of the reduction
movdqa %%T2, %%T3
pclmulqdq %%T2, %%GH, 0x00
psrldq  %%T2, 4

pclmulqdq %%GH, %%T3, 0x01
pslldq %%GH, 4

pxor %%GH, %%T2
pxor %%GH, %%T1
```

The optimization of Gue2013 is a little further described by Gueron as a Montgomery reduction but still lacks key elements explained to be able to apply it with other parameters, especially another defining polynomial for a finite field. The author cites an article of 2012 for this optimization but it is not currently available to the best of our knowledge. The code comes along an equivalent pseudo-code which shows an important constant and the name of the algorithmic technique used.

```
vmovdqa    T3, [W]
```

```
vpclmulqdq T2, T3, T7, 0x01
vpshufd    T4, T7, 78
vpxor      T4, T4, T2
vpclmulqdq T2, T3, T4, 0x01
vpshufd    T4, T4, 78
vpxor      T4, T4, T2
vpxor      T1, T1, T4  ; result in T1
```

```
Montgomery reduction
Input: 256-bit operand [X3 : X2 : X1 : X0]
[A1 : A0] = X0 • 0xc200000000000000
[B1 : B0] = [X0 ⊕ A1 : X1 ⊕ A0]
[C1 : C0] = B0 • 0xc200000000000000
[D1 : D0] = [B0 ⊕ C1 : B1 ⊕ C0]
Output: [D1 ⊕ X3 : D0 ⊕ X2]
```

The reversing goal is to reconstruct the high-level algorithm applied, determine what the constant represents, prove the equivalence of the two aforementioned optimizations—Ozturk and Gopal's PCLMULQDQ-based reduction and Gueron's Montgomery reduction, deduce the value of POLY2 and make it possible to apply this optimization on another finite field.

Reversing

The very idea behind the new optimization is based on rewriting the modular multiplication directly on the straight inputs. We make use of Gueron's slides Gue2013 to infer most of the explanations. They provide a formula which looks like a Montgomery's multiplication (FastREDC, see Algorithm 2.7 in BZ2010) modulo the defining polynomial reflected.

Rewriting the Modular Multiplication in GCM on Straight Inputs

As it is a major source of mistakes, we recall first that a binary polynomial represented by a binary word of size w has degree *at most* $w-1$.

Let us consider the polynomial $A(X) = A_{127}X^{127} + A_{126}X^{126} + \cdots + A_0$. If we denote by $ref(A, w)$ the reflected polynomial of A on a binary word of size w, with $w \geq 128$ we have $ref(A, w)(X) = A_0X^{w-1} + A_1X^{w-2} + \cdots + A_{127}X^{w-1-127} = A(X^{-1})X^{w-1}$.

As we mentioned earlier, all Intel's implementations of GCM work on *reflected* inputs in order to comply with the standard, whose choice of bit numbering is contrary to customs. In that respect, the Intel's GCM way to multiply two inputs A and B is the following:

```
Intel's GCM modular multiplication
T1 = ref(A, 128)
T2 = ref(B, 128)
T3 = T1 • T2
return ref(T3, 128)
```

with $T_3(X) = T_1(X) \times T_2(X) \bmod P(X)$. There is an useful and easy property one can check on reflected inputs when working with the polynomial representation (it is straightforward

when one writes it):

$$\begin{aligned} ref(A, w_A) \times ref(B, w_B) &= ref(A \times B, w_A + w_B - 1) \\ &= ref(A \times B, w_A + w_B) \operatorname{div} X. \end{aligned}$$

To grasp the idea intuitively, consider for instance A and B reflected on 128-bit words. The degrees of $ref(A, 128)$ and $ref(B, 128)$ is at most 127. The degree of $ref(A, 128) \times ref(B, 128)$ is at most 254, which means it can be represented on a 255-bit word. If it is reflected on a 256-bit word, we need to right shift it to make it become a 255-bit word (and a 255-degree polynomial).

The goal of this section is to explain how to express $ref(T_3, 128)$ directly from A and B without using their reflected representations.

As we have $T_3(X) = T_1(X) \times T_2(X) \bmod P(X)$, we will work on $T_1(X) \times T_2(X)$ to find properties to relate $ref(T_3, 128)$ directly to A and B.

- The Euclidean division rule allows us to write $T_1(X) \times T_2(X) = q(X) \times P(X) + T_3(X)$ with $deg(T_3) < 128$. As the degree of $T_1(X) \times T_2(X)$ is at most 254 and the degree of $P(X)$ is 128, we have $deg(q) \leq 126$.
- The property on the multiplication of reflected inputs allows us to write: $ref(T_1 \times T_2, 256) = ref(T_1, 128) \times ref(T_2, 128) \times X$.
- One can see that the reflection operation is a linear one, then $ref(T_1 \times T_2, 256) = ref(q \times P, 256) + ref(T_3, 256)$.
- By using our knowledge on the degrees of $q \times P$ and T_3, we are able to write:

  > - $ref(q \times P, 256) = ref(q, 127) \times ref(P, 129) \times X \equiv 0 \pmod{ref(P, 129)}$. We will denote in the sequel $p = ref(P, 129)$.
  > - $ref(T_3, 256) = ref(T_3, 128) \times X^{128}$.

We deduce from the above properties that $ref(T_3, 128) \times X^{128} \pmod p = A \times B \times X \pmod p$, which reconstructs the reasoning behind Gueron's equality in Gue2013:

$$ref(T_3, 128) = A \times (B \times X) \times X^{-128} \pmod p.$$

Recognizing Montgomery Fast Reduction

Montgomery reduction (either on integers or on polynomials) is a necessary step in the multiplication using elements in *Montgomery form*. When working with a fixed modulus n, instead of using a residue $a \bmod n$, one uses a residue $a' = \lambda a \bmod n$ with $\gcd(\lambda, n) = 1$. All operations are made in the *Montgomery domain* before putting back the result in its usual form by dividing it by λ. With a clever choice of λ, the division can be a shift. Addition of elements in Montgomery form is unchanged as $\lambda a + \lambda b = \lambda(a + b) \pmod n$. The multiplication of elements in Montgomery form is a little trickier as $\lambda a \times \lambda b = \lambda^2(ab) \pmod n$. To keep the result of a multiplication in the Montgomery form, one needs to compute efficiently the modular division by λ; it is the Montgomery reduction algorithm known as REDC.

$$\mathrm{REDC}(c) = \frac{c}{\lambda} \pmod{n}.$$

The variant known as FastREDC is of particular interest to us. In our case, we work on binary polynomials (hence beware of hasty generalizations, e.g. we removed all signs), $\lambda = X^{128}$ and the modulus we consider is $p(X) = X^{128} + X^{127} + X^{126} + X^{121} + 1$.

Require:

- X^{128}, the Montgomery basis
- U, the input polynomial to reduce, of degree at most 255
- p, the reduction polynomial of degree 128
- $p' = p^{-1} \pmod{X^{128}}$

Ensure: $T = U/X^{128} \pmod{p}$

1: **function** FastREDC(U, X^{128}, p, p')
2: $Q \leftarrow \left((U \bmod X^{128}) \times p'\right) \bmod X^{128}$
3: $T \leftarrow (U + Q \times p) \operatorname{div} X^{128}$
4: **return** T
5: **end function**

To recognize the Montgomery reduction in the optimization proposed in Gue2013, we have to notice that $p(X) = X^{128} + \texttt{0xc2}(0^{14})(X) \times X^{64} + 1$, where $\texttt{0xc2}(0^{14})(X)$ stands for the polynomial of degree 63 whose sequence of coefficients is `0xc2(0^14)`. We also have that $p' = p^{-1} \pmod{X^{128}} = \texttt{0xc2}(0^{14})(X)X^{64} + 1$. The fact that p and p' uses the same constant `0xc2(0^14)` forces us to understand finely the optimization if we are to apply it with other parameters.

Doing the computation *by hand* to check that the given algorithm corresponds to the optimization proposed in Gue2013 is left as an exercise to the reader.

Equivalence between Optimizations

If we make the assumption that Ozturk and Gopal's PCLMULQDQ-based reduction and Gueron's Montgomery reduction should rely on the same principle implemented slightly differently, we can simultaneously prove the equivalence between the two optimizations and deduce the value of `POLY2`.

In fact, by fixing `POLY2` to be the constant `0xC2(0^21)1C2(0^6)`, both computations end up being a Montgomery reduction, trading two `vpshufd` instructions against one `pclumulqdq`.

The trick to find the value of `POLY2` was to check in one Intel's GCM reference implementation, but it can also be painfully reconstructed by hand with the equivalence assumption in mind.

Again, doing the computation *by hand* is left as an exercise to the reader.

Putting Things Together

If we refer to BZ2010, Barrett's and Montgomery's algorithms are MSB and LSB variants of a division algorithm. The former is the *classical* division algorithm where one wants to cancel most significant digits and find the remainder; in the latter, one wants to cancel least significant digits and keep the most significant ones. Therefore, they naturally correspond to each other respectively on reflected and straight inputs.

We can see that the first so-called Barret's optimization allows easily to substitute `pclmulqdq` instructions by XORs and shifts. It is not the case anymore with the so-called Montgomery's optimization as computations involve modular multiplications with high degree polynomials. This can explain why trying to work directly with reflected inputs was not the priority in times when `pclmulqdq` was rather slow.

Applying It with Different Parameters

As an example, because we now can do it, let us apply the optimization on another finite field. If we were to do it on a smaller finite field, let's say the finite field with 2^{64} elements, how would we proceed?

We can use the polynomial defining the finite field with 2^{64} elements that was proposed as a complement in the original proposal of GCM: $P(X) = X^{64} + X^4 + X^3 + X + 1$. Its reflected is $p(X) = X^{64} + X^{63} + X^{61} + X^{60} + 1$ and $p' = p^{-1} \pmod{X^{64}} = X^{63} + X^{61} + X^{60} + 1$. We can write: $p(X) = X^{64} + \texttt{0xb0}^{15}(X) + 1$ and $p'(X) = \texttt{0xb0}^{15}(X) + 1$, which is still another case where the two polynomials make the same constant appear. One has to replace X^{128} in the pseudo-code of the function `FastREDC` by X^{64} and the input to it is of degree at most 127.

Applying the same optimization leads to the following piece of code:

```
Input: 128-bit operand [X1 : X0]
[A1 : A0] = X0 • 0xb000000000000000
[B1 : B0] = A0 • 0xb000000000000000
Output: [A0 ⊕ A1 ⊕ B1 ⊕ X0 ⊕ X1]
```

Conclusion

At this point we know that the piece of code:

```
vmovdqa    T3, [W]
vpclmulqdq T2, T3, T7, 0x01
vpshufd    T4, T7, 78
vpxor      T4, T4, T2
vpclmulqdq T2, T3, T4, 0x01
vpshufd    T4, T4, 78
vpxor      T4, T4, T2
vpxor      T1, T1, T4  ; result in T1
```

does a Montgomery reduction on a binary polynomial of degree at most 255, a basis X^{128} and the modulus $p(X) = X^{128} + X^{127} + X^{126} + X^{121} + 1$. Most importantly, we understand how and why, enough to be able to reconstruct the high level algorithm and apply it with different parameters.

Acknowledgments

Marion Videau would like to thank Paul Zimmermann for taking time explaining computer arithmetic both in real life and in books and Pierrick Gaudry for being a supportive and compassionate office mate when she worked on this topic.

References

[1] https://en.wikipedia.org/wiki/AES_instruction_set

[2] https://en.wikipedia.org/wiki/CLMUL_instruction_set

[3] https://en.wikipedia.org/wiki/Universal_hashing

[GCM2007] NIST Special Publication 800-38D. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, November 2007. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf Original proposal to NIST from David A. McGrew and John Viega (2004).

[AES2001] NIST Federal Information Processing Standards Publication 197. *Announcing the Advanced Encryption Standard (AES)*, November 26, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf Original proposal to NIST from Joan Daemen and Vincent Rijmen (1998).

[Rog2011] Philip Rogaway. *Evaluation of Some Blockcipher Modes of Operation*, Evaluation carried out for the Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan, Feb. 2011. http://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf

[Fog2016] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, version 2016-Jan-09. http://www.agner.org/optimize/instruction_tables.pdf

[Gue2013] Shay Gueron. *AES-GCM for Efficient Authenticated Encryption - Ending the reign of HMAC-SHA-1?*, Workshop on Real-World Cryptography, 2013. https://crypto.stanford.edu/RealWorldCrypto/slides/gueron.pdf

[GK2010] Shay Gueron and Michael Kounavis. *Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm*, Information Processing Letters 110 (2010) 549-553. http://dx.doi.org/10.1016/j.ipl.2010.04.011

[OG2012] Erdinc Ozturk and Vinodh Gopal. *Enabling High-Performance Galois-Counter-Mode on Intel Architecture Processors*, Oct. 2012, Intel. http://www.intel.com/content/dam/www/public/us/en/documents/software-support/enabling-high-performance-gcm.pdf

[BZ2010] Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic*, Cambridge University Press, 2010. http://www.loria.fr/~zimmerma/mca/mca-cup-0.5.9.pdf

Comments