TU Graz

SCIENCE
PASSION
TECHNOLOGY

Practical Use of Finite Fields and their Performance on Modern CPUs

GHASH in AES-GCM

Daniel Kales

October 31st , 2019

Outline

1 AES-GCM

2 GHASH

3 Finite Field Multiplication on x86 CPUs

4 Performance

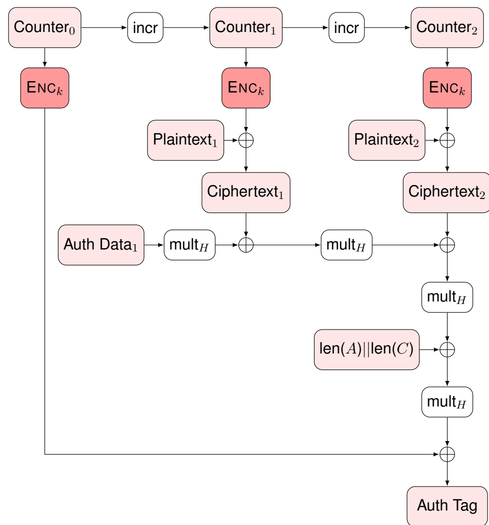
AES-GCM

Introduction

- Authenticated Encryption (AE)
  - Very important building block in TLS
  - TLS 1.3: Only AE modes allowed
- AE modes in TLS 1.3
  - AES-{128,256}-GCM
  - ChaCha20-Poly1305
  - AES-128-CCM

AES-GCM

- AES in Galois Counter Mode
- Combination of:
  - AES in Counter Mode
  - GHASH authenticator



GHASH

Universal Hash Functions

- Concept by Carter and Wegman in 1977[1]
  - Family of hash funtions $H = \{h : U \mapsto [m]\}$ is called universal if

$$\forall x, y \in U, x \neq y : \Pr_{h \in H}[h(x) = h(y)] = \frac{1}{m}$$

  - In other words, collision probability is as low as if hash values are truly randomly assigned for each key.
  - Cyptographic properties such as preimage resistance not required!

[1] Larry Carter and Mark N. Wegman, Universal Classes of Hash Functions, Proceedings of the 9th Annual ACM Symposium on Theory of Computing, 1977

Authenticators from Universal Hash Functions

Building a message authentication code (MAC) from universal hash functions:

1. Use universal hash function (selected by secret key) and hash message to short digest
2. Encrypt short digest by adding a one-time key

Provably secure in the information theoretic setting!

- one-time key not good for usability
  - replaced by function of nonce and secret key

GHASH

Polynomial based on input message blocks S_i (in AES-GCM this is the ciphertext), evaluated at secret key $H = \text{AES}_k(0^{128})$

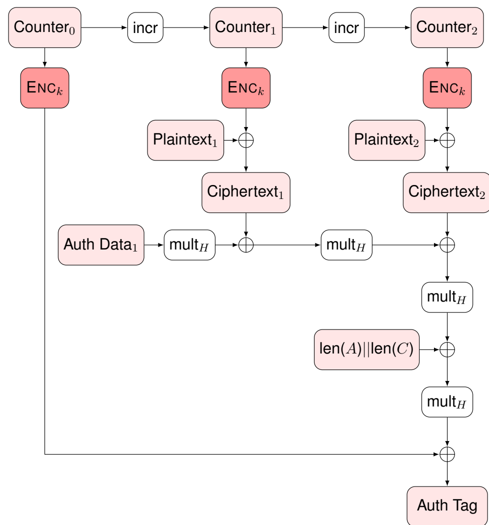
$$\text{GHASH}(H, A, C) = X_{m+n+1}$$

$$X_i = \sum_{j=1}^{i} S_j \cdot H^{i-j+1} = \begin{cases} 0 & \text{for } i = 0 \\ (X_{i-1} + S_i) \cdot H & \text{otherwise.} \end{cases}$$

Second form is iterative and used in most cases.

AES-GCM

- Look at structure of GHASH
  - Iterative form
  - final "one-time" key addition



Finite Field Multiplication on x86 CPUs

PCLMULQDQ

- Specialized CPU instruction
- **C**arry**l**ess **Mul**tiplication
  - 64-bit $\times$ 64-bit $\mapsto$ 128-bit
- Carryless multiplication is equivalent to multiplication of polynomials over field GF(2)
  - GF(2^n) is usually represented as polynomials in GF$(2)[X]$
  - Finite field multiplication is polynomial multiplication
    - followed by reduction with respect to some irreducible polynomial

PCLMULQDQ (cont.)

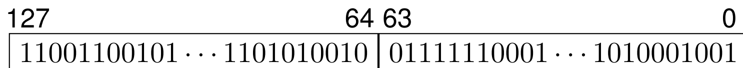
- Representation of polynomial coefficients as bitstring:

$$a_{n-1}X^{n-1} + \cdots + a_1X + a_0 \leftrightarrow a_{n-1}||\cdots||a_1||a_0$$

- Example:

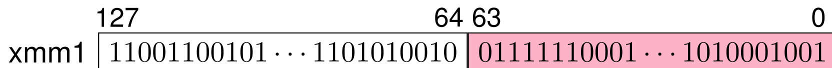
$$X^7 + X^3 + X + 1 \leftrightarrow 10001011$$

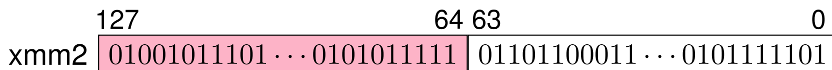
- Store one GF(2^{128}) element in a 128-bit CPU register



PCLMULQDQ (cont.)

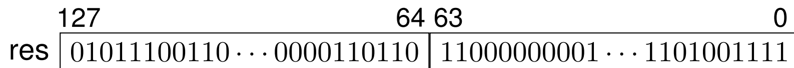
- PCLMULQDQ takes two 128-bit registers and a selection value imm





imm 10

res = xmm1[0:63] × xmm2[127:64]



Multiplication of GF(2^{128}) elements

Combine the result of 4 sub-multiplications

	127 … 64	63 … 0
xmm1	$11001100101 \cdots 1101010010$	$01111110001 \cdots 1010001001$
xmm2	$01001011101 \cdots 0101011111$	$01101100011 \cdots 0101111101$

	255 … 192	191 … 128	127 … 64	63 … 0
	$01011100110 \cdots 0000110110$	$11000000001 \cdots 1101001111$		
		$01011100110 \cdots 0000110110$	$11000000001 \cdots 1101001111$	
		$01011100110 \cdots 0000110110$	$11000000001 \cdots 1101001111$	
			$01011100110 \cdots 0000110110$	$11000000001 \cdots 1101001111$
Result	$11000000001 \cdots 1101001111$	$11000000001 \cdots 1101001111$	$11000000001 \cdots 1101001111$	$11000000001 \cdots 1101001111$

Modular Reduction

- Result needs to be reduced modulo the irreducible polynomial
  - No specialized instruction
- Option 1: Standard long division
  - Slow and tedious to implement
- Option 2: efficient reduction algorithm
  - Realizes division using only 2 multiplications
  - Special form of Barret reduction algorithm[2]

[2] P. Barrett, Implementing the Rivest, Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor, Master's Thesis, University of Oxford, UK, 1986

Efficient Modular Reduction

- We have a 256-bit polynomial $a(X)$ and want to reduce it modulo $g(X)$:

$$r(X) = a(X) \text{ mod } g(X)\,.$$

- use linearity to split calculation in two halves:

$$\begin{aligned} r(X) &= \big(c(X) \cdot X^{128} + b(X)\big) \text{ mod } p(X) = c(X) \cdot X^{128} \text{ mod } p(X) + b(X) \text{ mod } p(X) \\ &= c(X) \cdot X^{128} \text{ mod } p(X) + b(X)\,. \end{aligned}$$

- Focus on effcient calculation of $p(X) = c(X) \cdot X^{128} \text{ mod } p(X)$
  - “Reduce upper half of 256-bit register and xor result with lower half”

Efficient Modular Reduction (cont.)

$$p(X) = c(X) \cdot X^{128} \text{ mod } g(X) = g(X) \cdot q(X) \text{ mod } X^{128},$$

where $q(X)$ is the result of the division of $c(X) \cdot X^{128}$ by $g(X)$:

$$c(X) \cdot X^{128} = g(X) \cdot q(X) + p(X).$$

Why does the first equality hold?

Since the 128 least significant terms of $c(X) \cdot X^{128}$ are zero, the least significant 128 bits of $g(X) \cdot q(X)$ and $p(X)$ must be equal, so they cancel to zero in the addition. This reduces the modular reduction to finding the quotient $q(X)$ and a finite field multiplication (the final reduction modulo X^{128} is equivalent to taking the lower 128 bit of the result).

Efficient Modular Reduction (cont.)

$$p(X) = c(X) \cdot X^{128} \bmod g(X) = g(X) \cdot q(X) \bmod X^{128},$$

where $q(X)$ is the result of the division of $c(X) \cdot X^{128}$ by $g(X)$:

$$c(X) \cdot X^{128} = g(X) \cdot q(X) + p(X).$$

Why does the first equality hold?
Since the 128 least significant terms of $c(X) \cdot X^{128}$ are zero, the least significant 128 bits of $g(X) \cdot q(X)$ and $p(X)$ must be equal, so they cancel to zero in the addition. This reduces the modular reduction to finding the quotient $q(X)$ and a finite field multiplication (the final reduction modulo X^{128} is equivalent to taking the lower 128 bit of the result).

Efficient Modular Reduction (cont.)

How to find $q(X)$?

$$q(X) = MSB\big(c(X) \cdot q^{+}(X)\big)\,,$$

where $q^{+}(X)$ is the result of the division of X^{256} by $g(X)$ (precompute once):

$$X^{256} = g(X) \cdot q^{+}(X) + p^{+}(X)\,.$$

This is similar to the calculation of the quotient in the Barrett reduction algorithm. For more details on this step, refer to the Intel White Paper[3].

[3] https://software.intel.com/sites/default/files/managed/72/cc/clmul-wp-rev-2.02-2014-04-20.pdf

Efficient Modular Reduction Algorithm

- Precompute $q^+(X)$ for the given irreducible polynomial $g(X)$

Modular reduction algorithm:

1. Multiply $c(X)$, the upper half of the input, with $q^+(X)$
2. Take the upper 128-bit half of the result and multiply it with $g(X)$
3. Add $b(X)$, the lower half of the input to the result of the calculation
4. Return the 128 least significant bits as the reduced result

Performance

Performance of AES-GCM

Features	Throughput [MB/s]
(CT) Software only	67.24
AES-NI	224.20
PCLMULQDQ	87.73
Both	1013.63

Table: Performance of AES-128-GCM on Intel Xeon E3-1220 @ 3.1GHz[4]

Without specialized instructions for PCLMULQDQ, the calculation of GHASH bottlenecks the AES computations with AES-NI.

[4] Based on Thomas Pornin's BearSSL benchmarks: https://www.bearssl.org/speed.html

Questions?