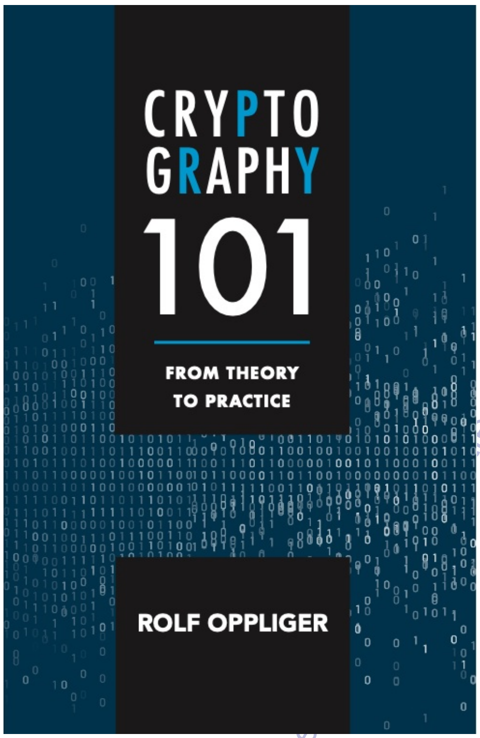

This text is an updated version of Section 11.2.2 from a book written by Rolf Oppliger in 2021 (ISBN 978-1-63081-846-3)

11.2.2 GCM

Like CCM, GCM is designed to use a 128-bit block cipher (e.g., AES) in CTR mode. But unlike CCM, the CTR mode of GCM uses a unique counter incrementation function and a message authentication construction that employs a universal hash function based on polynomial evaluation in $GF\ (2^{128})$. According to Section 10.3.3, this construction yields a Carter-Wegman MAC. The NIST document that specifies the GCM mode [6] also refers to an authentication-only variant of GCM called Ga-lois message authentication code (GMAC). In short, GMAC uses GCM encryption but requires no data to be encrypted, meaning that all data are only authenticated.

$GF(2^{128})$ is an extension field of $GF(2)$. Its elements are strings of 128 bits, and its operations are addition ($\oplus$) and multiplication ($\cdot$). If $x = x_0x_1 \ldots x_{127}$ and $y = y_0y_1 \ldots y_{127}$ are two elements of $GF(2^{128})$ with x_i and y_i representing bits for $i = 0, 1, \ldots, 127$, then $x \oplus y$ can be implemented as bitwise addition modulo 2 and $x \cdot y$ can be implemented as polynomial multiplication modulo an irreducible polynomial (Appendix A.3.6). In the case of GCM, this polynomial is set to $f(x) = 1 + x + x^2 + x^7 + x^{128}$ (according to the standard).

Algorithm 11.3 GHASH function used in GCM mode.

```
(h, x)
---
y_0 = 0^128
for i = 1 to n do y_i = (y_{i-1} ⊕ x_i) · h
---
(y_n)
```

GCM employs two complementary functions: A hash function called GHASH and an encryption function called GCTR that is a variant of "normal" CTR mode encryption.

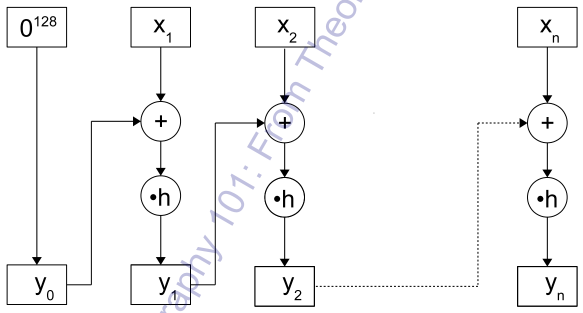

Figure 11.2 The GHASH function.

- The GHASH function is specified in Algorithm 11.3 and illustrated in Figure 11.2.[2] It takes as input a 128-bit hash subkey h and $x = x_1 \parallel x_2 \parallel \ldots \parallel x_n$ that is a sequence of n 128-bit blocks $x_1, x_2, \ldots, x_n$, and it generates as output a 128-bit hash value y_n. The function is simple and straightforward: It starts with a 128-bit block y_0 that is initialized with 128 zeros (written as 0^{128}), and it then iteratively adds the next block of x and multiplies the result with the hash subkey h. This is iterated n times, until y_n is returned as output. More specifically, $y_1, y_2, \ldots, y_n$ can be computed as follows:

2 While GHASH is a hash function, it is not a cryptographic one.

$$\begin{aligned}
y_1 &= (y_0 \oplus x_1) \cdot h = x_1 \cdot h \\
y_2 &= (y_1 \oplus x_2) \cdot h = (((y_0 \oplus x_1) \cdot h) \oplus x_2) \cdot h = x_1 \cdot h^2 \oplus x_2 \cdot h \\
y_3 &= (y_2 \oplus x_3) \cdot h = ((x_1 \cdot h^2 \oplus x_2 \cdot h) \oplus x_3) \cdot h \\
&= x_1 \cdot h^3 \oplus x_2 \cdot h^2 \oplus x_3 \cdot h \\
&\dots \\
y_n &= x_1 \cdot h^n \oplus x_2 \cdot h^{n-1} \oplus \ \dots \ \oplus x_{n-1} \cdot h^2 \oplus x_n \cdot h \\
&= \bigoplus_{i=1}^{n} x_i \cdot h^{n+1-i}
\end{aligned}$$

Note that y_0 does not influence the computation, and that only the last value y_n is needed to form the output (all other values are noted here just for the sake of transparency).

Algorithm 11.4 GCTR encryption function.

```
(k,ICB, x)
------------------------------
if $x$ is empty then return empty bit string $y$
$n = \lceil |x|/128 \rceil$
$b_1 =$ ICB
for $i = 2$ to $n$ do $b_i = \text{inc}_{32}(b_{i-1})$
for $i = 1$ to $n-1$ do $y_i = x_i \oplus E_k(b_i)$
$y_n = x_n \oplus \text{MSB}_{|x_n|}(E_k(b_n))$
$y = y_1 \parallel y_2 \parallel \dots \parallel y_{n-1} \parallel y_n$
------------------------------
(y)
```

- The GCTR encryption function is specified in Algorithm 11.4. It takes as input a key k, an initial counter block (ICB), and an arbitrarily long bit string x, and it generates as output another bit string y that represents the encrypted version of x using k and the ICB. More specifically, $x = x_1 \parallel x_2 \parallel \dots \parallel x_n$ is a sequence of $n = \lceil |x|/128 \rceil$ blocks, where $x_1, x_2, \dots, x_{n-1}$ are complete 128-bit blocks but x_n does not need to be complete (i.e., $|x_n| \leq 128$). The algorithm uses a sequence of n 128-bit counter blocks $b_1, b_2, \dots, b_n$ that are encrypted and then added modulo 2 to the respective blocks of x. If x_n is incomplete, then the respective number of b_n's most significant bits are used and the remaining bits of b_n are simply discarded. In the end, y is compiled as the concatenation of all n ciphertext blocks $y_1, y_2, \dots, y_n$, where y_n can again be incomplete. The auxiliary function $\text{inc}_s(\cdot)$ increments the least significant

s bits of a block and leaves the remaining $128 - s$ bits unchanged. This can be formally expressed as follows:

$$\text{inc}_s(x) = \text{MSB}_{128-s}(x) \parallel [\text{int}(\text{LSB}_s(x)) + 1 \pmod{2^s}]_s$$

In GCTR and GCM, s is 32 bits. This means that the first 96 bits of x remain unchanged and only the last 32 bits are incremented in each step.

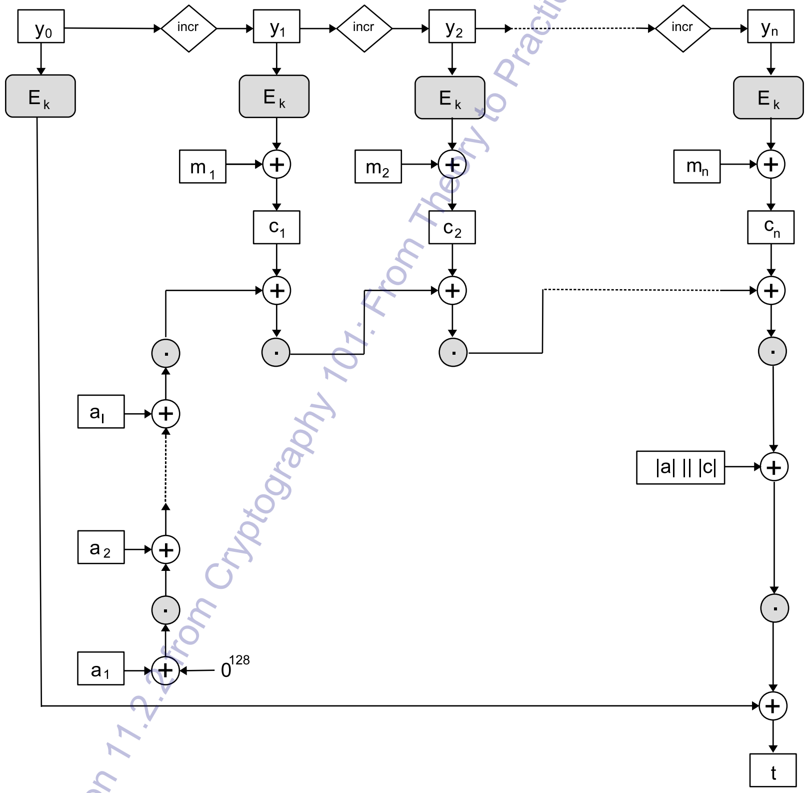

Figure 11.3 GCM authenticated encryption.

Having prepared all the ingredients, we are now ready to explain and delve more deeply into GCM authenticated encryption as specified in Algorithm 11.5 and partly illustrated in Figure 11.3. As is usually the case in AEAD, the algorithm takes as input a key k for the block cipher in use, a variable-length nonce r,[3] a message m to be encrypted and authenticated, and some additional data a to be authenticated. The message m comprises $0 < |m| \leq 2^{39} - 256$ bits that form n 128-bit blocks, whereas a is $0 < |a| \leq 2^{64} - 1$ bits long and forms $l = \lceil |a|/128 \rceil$ 128-bit blocks. The output of the algorithm consists of two parts: The ciphertext c that is equally long as m and the authentication tag t that can have a variable length (up to $2^{64} - 1$ bits). The official specification suggests that t is 128, 120, 112, 104, or 96 bits long, and that there may be some exceptional applications and use cases where t is only 64 or 32 bits long.

Algorithm 11.5 GCM authenticated encryption.

```
$(k, r, m, a)$
----------------------------------------
$h = E_k(0^{128})$
if $|r| = 96$ then $y_0 = r \| 0^{31}1$
            else $s = 128 \cdot \lceil |r|/128 \rceil - |r|$
                 $y_0 = \text{GHASH}(h, (r \| 0^{s+64} \| [|r|]_{64}))$
$c = \text{GCTR}(k, \text{inc}_{32}(y_0), m)$
$pad_a = 128 \cdot \lceil |a|/128 \rceil - |a|$
$pad_c = 128 \cdot \lceil |c|/128 \rceil - |c|$
$b = \text{GHASH}(h, (a \| 0^{pad_a} \| c \| 0^{pad_c} \| [|a|]_{64} \| [|c|]_{64}))$
$t = \text{MSB}_{|t|}(\text{GCTR}(k, y_0, b)$
----------------------------------------
$(c, t)$
```

The GCM encryption algorithm first generates a subkey h for the GHASH function. This value is generated by encrypting a block that consists of 128 zero bits (i.e., 0^{128}) with the block cipher and key k. It then derives a 128-bit precounter block y_0 from the variable-length nonce r. This derivation is formally expressed in Algorithm 11.5 but is not illustrated in Figure 11.3. In the most likely case that r is 96 bits long, y_0 is just the concatenation of r, 31 zero bits, and a one bit. This yields 128 bits in total. If, however, r is not 96 bits long, then the construction of y_0 is slightly more involved. In this case, r is padded some with some zero bits so that the concatenation of r, the zero bits, and the 64-bit length encoding of r yields a string that is a multiple of 128 bits long (in Algorithm 11.5 a temporary variable s is used to determine the number of zeros). This string is then subject to the GHASH function with subkey h, so that the resulting precounter block y_0 is again 128 bits long. In either case, this is the value Figure 11.3 starts with in the upper left corner. The algorithm generates a sequence of counter values from y_0 by

3 Again, the distinction between a nonce and an IV is somehow vague. While the GCM specification uses the notion of an IV, we use the notion of a nonce. It is particularly important that the value does not repeat, and this is best characterized with the notion of a nonce.

recursively applying the 32-bit incrementing function $\text{inc}_{32}(\cdot)$. All y values except y_0 are then used to GCTR-encrypt the message m with the key k. This yields the ciphertext $c = c_1 \parallel c_2 \parallel \ldots \parallel c_n$ and terminates the encryption part of the algorithm.

The second part of the algorithm deals with authentication and generates a respective tag t. The algorithm therefore computes the minimum numbers of zero bits, possibly none, to pad a and c so that the bit lengths of the respective strings are both multiples of 128 bits. The resulting values are pad_a for a and pad_c for c. The algorithm then pads a and c with the appropriate number of zeros, so that the concatenation of a, 0^{pad_a}, c, and 0^{pad_c}, as well as the 64-bit length representations of a and c is a multiple of 128 bits long. Again, this string is subject to the GHASH function with subkey h. The result is b and this 128-bit string is input to the GCTR function—together with the key k and the formerly generated precounter block y_0. If the tag length is $|t|$, then t refers to the $|t|$ leftmost (most significant) bits of the output of the GCTR function. The output of the GCM authenticated encryption algorithm consists of c and t.

Algorithm 11.6 GCM authenticated decryption.

```
$(k, r, c, a, t)$
-----------------------------------------------
verify lengths of $r$, $c$, $a$, and $t$
$h = E_k(0^{128})$
if $|r| = 96$ then $y_0 = r \parallel 0^{31}1$
              else $s = 128 \cdot \lceil |r|/128 \rceil - |r|$
                   $y_0 = \text{GHASH}(h, (r \parallel 0^{s+64} \parallel [|r|]_{64}))$
$m = \text{GCTR}(k, \text{inc}_{32}(y_0), c)$
$pad_c = 128 \cdot \lceil |c|/128 \rceil - |c|$
$pad_a = 128 \cdot \lceil |a|/128 \rceil - |a|$
$b = \text{GHASH}(h, (a \parallel 0^{pad_a} \parallel c \parallel 0^{pad_c} \parallel [|a|]_{64} \parallel [|c|]_{64}))$
$t' = \text{MSB}_{|t|}(\text{GCTR}(k, y_0, b)$
if $t = t'$ then return $m$ else return **FAIL**
-----------------------------------------------
($m$ or **FAIL**)
```

GCM authenticated decryption works similarly, but the operations are performed in more or less reverse order. It is specified in Algorithm 11.6 and partly illustrated in Figure 11.4. The algorithm takes as input k and r that are the same as used for encryption, as well as c, a, and t, and it generates as output either m or **FAIL**. First, the algorithm verifies the lengths of r, c, a, and t. If at least one of these lengths is invalid, then the algorithm aborts and returns **FAIL** (this is not explicitly mentioned in Algorithm 11.6). Next, the algorithm generates the subkey h (i.e., it therefore encrypts the zero block with the block cipher and the key k) and the precounter block y_0 in exactly the same way as before. In the next step,

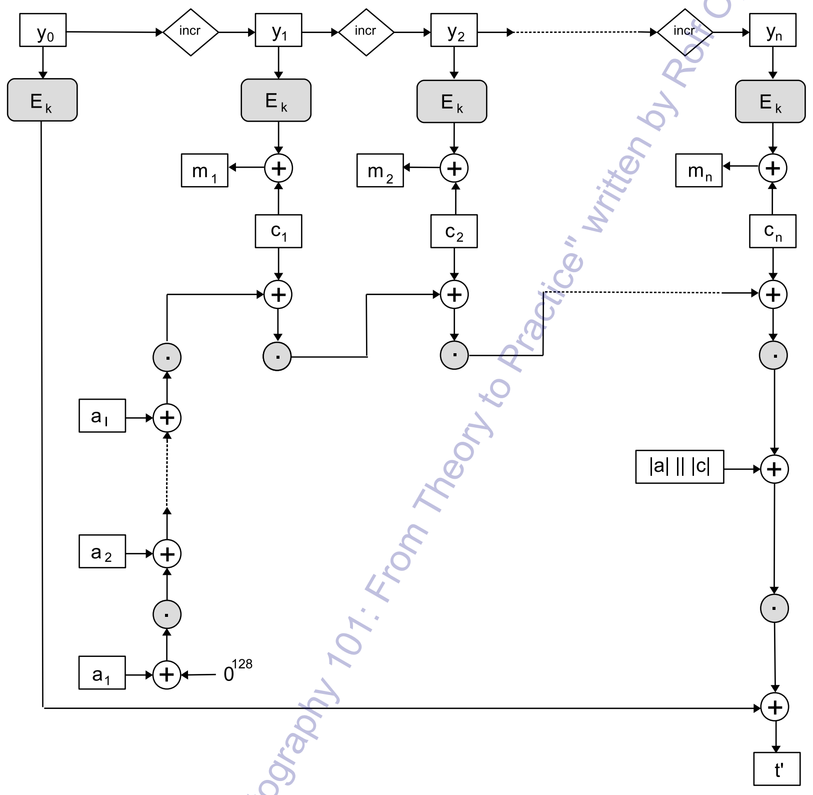

Figure 11.4 GCM authenticated decryption.

the message m is decrypted. This step is essentially the same as in the encryption algorithm, except that the roles of m and c are swapped. Also, the computations of pad_a and pad_c as well as b are identical. On the basis of b, the authentication tag t' can be recomputed. Decryption is successful if and only if t' equals the originally received value t. Otherwise, decryption fails and the algorithm signals this fact by returning **FAIL**. Note that the final verification step, i.e., verify whether t' equals t, is not illustrated in Figure 11.4.

It is commonly believed that the GCM mode is secure as long as a new and fresh nonce r is used for every single message. If a nonce is reused, then it may become feasible to learn the authentication key (i.e., the hash subkey h) and to use it to forge authentication tags. Because the encryption is a stream cipher that is highly malleable, the ability to forge authentication tags may empower an adversary to mount some sophisticated attacks. Unfortunately, some Internet security protocol specifications do not clearly specify how to generate nonces in a secure way. For example, the specification of the TLS protocol does not say anything about the generation of nonces for AES-GCM. Consequently, there are a few insecure implementations that reuse nonces.[4] Needless to say, these implementations are susceptible to cryptanalysis and do not provide the level of security that is otherwise anticipated with AES-GCM.

The uniqueness requirement for the nonce is addressed in Section 8 of the standard [6], together with two techniques to construct respective nonces: A deterministic construction and an RBG-based construction that both use two fields to come up with a nonce that is as unique as possible. Furthermore, people have also developed nonce-misuse resistant AE and AEAD modes for block ciphers, such as AES-SIV [7] or—more recently—AES-GCM-SIV [8].[5] In either case, the acronym SIV stands for synthetic IV, and the idea is to deterministically derive the nonce from the message (so that different messages automatically lead to different nonces). If one has the choice, then it is certainly a good idea to use this technology to make AE or AEAD modes nonce-misuse resistant.

References

[6] U.S. Department of Commerce, National Institute of Standards and Technology, Recommenda-tion for Block Cipher Modes of Operation: Galois/ Counter Mode (GCM) and GMAC, Special Publication 800-38D, November 2007.

[7] Harkins, D., Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES), RFC 5297, October 2008.