The Galois/Counter Mode of Operation (GCM)

David . McGrew
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, C 95032
mcgrew@cisco com

John Viega
Secure Software
4100 Lafayette Center Drive, Suite 100
Chantilly, V 20151
viega@securesoftware com

May 31, 2005

Contents

1 Introduction

Galois/Counter Mode (GCM) is a block cipher mode of operation that uses universal hashing over a binary Galois field to provide authenticated encryption. It can be implemented in hardware to achieve high speeds with low cost and low latency. Software implementations can achieve excellent performance by using table-driven field operations. It uses mechanisms that are supported by a well-understood theoretical foundation, and its security follows from a single reasonable assumption about the security of the block cipher.

There is a compelling need for a mode of operation that can efficiently provide authenticated encryption at speeds of 10 gigabits per second and above in hardware, perform well in software, and is free of intellectual property restrictions. The mode must admit pipelined and paralellized implementations and have minimal computational latency in order to be useful at high data rates. Counter mode has emerged as the best method for high-speed encryption, because it meets those requirements. However, there is no suitable standard message authentication algorithm. This fact leaves us in the situation in which we can encrypt at high speed, but we cannot provide message authentication that can keep up with our cipher. This lack is especially conspicuous since counter mode provides no protection against bit-flipping attacks.

GCM fills this need, while no other proposed mode meets the same criteria. CBC-M C [1, p-pendix F] and the modes that use it to provide authentication, such as CCM [2], E X [3], and OM C [4], cannot be pipelined or parallelized, and thus are unsuitable for high data rates. OCB [5] is covered by multiple intellectual property claims. CWC [6] does not share those problems, but is less appropriate for high speed implementations. In particular, CWC's message authentication component uses 127-bit integer multiplication operations whose implementation costs exceed those of even ES counter mode at high speeds, and it has a circuit depth that is twice that of GCM. In contrast, the binary field multiplication used to provide authentication in GCM is easily implemented at a fraction of the cost of counter mode at high speeds.

GCM also has additional useful properties. It is capable of acting as a stand-alone M C, authenticating messages when there is no data to encrypt, with no modifications. Importantly, it can be used as an incremental M C [7]: if an authentication tag is computed for a message, then part of the message is changed, an authentication tag can be computed for the new message with computational cost proportional to the number of bits that were changed. This feature is unique among all of the proposed modes.

nother useful property is that it accepts initialization vectors of arbitrary length, which makes it easier for applications to meet the requirement that all IVs be distinct. In many situations in which authenticated encryption is needed, there is a data element that could be used as a nonce, or as a part of a nonce, except that the length of the element(s) may exceed the block size of the cipher. In GCM, a nonce of any size can be used as the IV. This property is shared with E X, but no other proposed mode.

This document is organized as follows. Section 2 contains a complete specification of GCM, and is the only normative part of this document. Section 3 contains an overview of finite fields and a detailed description of the field representation used in GCM. Implementation strategies are described in Section 4, along with a discussion of their performance. summary of the mode's properties and a rationale for its design is offered in Section 6, along with a detailed performance comparison with other modes. The security analysis is summarized in Section 7. ppendix describes the use of GCM for 64-bit block ciphers. Test data that can be used for validating ES GCM implementations is contained in ppendix B.

This is the second version of the GCM specification. It makes no normative changes from the initial version posted on the NIST Modes of Operation web site on January 15, 2004, except for added guidance on the authentication tag length and other parameters, which is now provided at the end of Section 2.1. It corrects a number of minor errors, none of which were in the normative definition of ES GCM, and makes some clarifications. Most importantly, lgorithm 2 in the section on implementation has been corrected, as has the 64-bit IV definition in ppendix . We would like to thank the many reviewers whose comments and discussions have benefitted GCM and improved our work. We want to especially acknowledge Mike Boyle, Niels Ferguson, Scott Fluhrer, Brian Gladman, Russ Housley, wais Nemat, Chris Salter, and Colin Sinclair.

2 Definition

This section contains the complete definition of GCM for 128-bit block ciphers. The mode is slightly different when applied to 64-bit block ciphers; those differences are outlined in ppendix .

2.1 Inputs and Outputs

GCM has two operations, authenticated encryption and authenticated decryption. The authenticated encryption operation has four inputs, each of which is a bit string:

- secret key K, whose length is appropriate for the underlying block cipher.
- n initialization vector IV, that can have any number of bits between 1 and 2^{64}. For a fixed value of the key, each IV value must be distinct, but need not have equal lengths. 96-bit IV values can be processed more efficiently, so that length is recommended for situations in which efficiency is critical.

- plaintext P, which can have any number of bits between 0 and 2^{39} 256.
- dditional authenticated data (D), which is denoted as A. This data is authenticated, but not encrypted, and can have any number of bits between 0 and 2^{64}.

There are two outputs:

- ciphertext C whose length is exactly that of the plaintext P.
- n authentication tag T, whose length can be any value between 64 and 128. The length of the tag is denoted as t, and guidance on suitable values is provided in Section .

The inputs and outputs are defined in terms of bit strings. However, an implementation may choose to accept only inputs that are 8-bit aligned, that is, it may accept only byte strings. n implementation may similarly restrict the tag size.

The authenticated decryption operation has five inputs: K, IV, C, A, and T. It has only a single output, either the plaintext value P or a special symbol **F IL** that indicates that the inputs are not authentic. ciphertext C, initialization vector IV, additional authenticated data A and tag T are authentic for key K when they are generated by the encrypt operation with inputs K, IV, A and P, for some plaintext P. The authenticated decrypt operation will, with high probability, return **F IL** whenever its inputs were not created by the encrypt operation with the identical key.

The additional authenticated data A is used to protect information that needs to be authenticated, but which must be left unencrypted. When using GCM to secure a network protocol, this input could include addresses, ports, sequence numbers, protocol version numbers, and other fields that indicate how the plaintext should be handled, forwarded, or processed. In many situations, it is desirable to authenticate these fields, though they must be left in the clear to allow the network or system to function properly. When this data is included in the D, authentication is provided without copying the data into the ciphertext.

The primary purpose of the IV is to be a nonce, that is, to be distinct for each invocation of the encryption operation for a fixed key. It is acceptable for the IV to be generated randomly, as long as the distinctness of the IV values for each key is highly likely. The IV is authenticated, and it is not necessary to include it in the D field.

Both confidentiality and message authentication is provided on the plaintext. The strength of the authentication of P, IV and A is determined by the length t of the authentication tag. When the length of P is zero, GCM acts as a M C on the input A. The mode of operation that uses GCM as a stand-alone message authentication code is denoted as GM C.

The tag length t must be fixed for any fixed value of the key, and must be at least 64 bits. tag length of 128 bits should be used whenever possible, because this value provides the best security[1]. If an IV with a length other than 96 bits is used with a particular key, then that key must be used with a tag length of 128 bits.

n attacker can attempt to forge a t-bit M C for a message by choosing it at random. This attack will succeed with probability $2^{\ t}$. With GCM, an attacker is able to choose tags that have a probability of $B+1)2^{\ t}$ of succeeding, where B is the number of 128-bit blocks in the plaintext and additional authenticated data. Thus the effective tag strength for GCM is about $t \quad \lg B$ bits.

s with most modes of operation, security degrades as more data is processed with a single key. The total length of the plaintext and additional associated data protected with a single key should not exceed 2^{68} bytes.

n example use of GCM for network security is provided in Section 5, which shows how the inputs and outputs can be used in a typical cryptographic application.

2.2 Notation

Our notation follows that of the *Recommendation for Block Cipher Modes of Operation* [8]. The two main functions used in GCM are block cipher encryption and multiplication over the field $GF\ 2^{128})$. The block cipher encryption of the value X with the key K is denoted as $E\ K, X)$. The multiplication of two elements $X, Y \in GF\ 2^{128})$ is denoted as $X \cdot Y$, and the addition of X and Y is denoted as $X \oplus Y$. ddition in this field is equivalent to the bitwise exclusive-or operation, and the multiplication operation is defined in Section 2.5.

The function len) returns a 64-bit string containing the nonnegative integer describing the number of bits in its argument, with the least significant bit on the right. The expression 0^l denotes a string of l zero bits, and $A \| B$ denotes the concatenation of two bit strings A and B. The function $\mathrm{MSB}_t\ S)$ returns the bit string containing only the most significant (leftmost) t bits of S, and the symbol $\{\}$ denotes the bit string with zero length.

2.3 Encryption

Let n and u denote the unique pair of positive integers such that the total number of bits in the plaintext is $n \quad 1)128 + u$, where $1 \leq u \leq 128$. The plaintext consists of a sequence of n bit

[1] For some message authentication codes, a slight reduction in the size of the tag improves resistance against certain attacks. This is not true for GCM.

strings, in which the bit length of the last bit string is u, and the bit length of the other bit strings is 128. The sequence is denoted $P_1, P_2, \ldots, P_{n \quad 1}, P_n^*$, and the bit strings are called data blocks, although the last bit string, P_n^* , may not be a complete block. Similarly, the ciphertext is denoted as $C_1, C_2, \ldots, C_{n \quad 1}, C_n^*$, where the number of bits in the final block C_n^* is u. The additional authenticated data A is denoted as $A_1, A_2, \ldots, A_{m \quad 1}, A_m^*$, where the last bit string A_m^* may be a partial block of length v, and m and v denote the unique pair of positive integers such that the total number of bits in A is $\quad m \quad 1)128 + v$ and $1 \leq v \leq 128$.

The authenticated encryption operation is defined by the following equations:

$$
\begin{aligned}
H &= E \;\; K, 0^{128}) \\
Y_0 &= \begin{cases} IV \| 0^{31}1 & \text{if len } \; IV) = 96 \\ \text{GH} \quad \text{SH} \;\; H, \{\}, IV) & \text{otherwise.} \end{cases} \\
Y_i &= \text{incr} \;\; Y_{i \quad 1}) \text{ for } i = 1, \ldots, n \\
C_i &= P_i \oplus E \;\; K, Y_i) \text{ for } i = 1, \ldots, n \quad 1 \\
C_n^* &= P_n^* \oplus \text{MSB}_u \;\; E \;\; K, Y_n)) \\
T &= \text{MSB}_t \;\; \text{GH} \quad \text{SH} \;\; H, A, C) \oplus E \;\; K, Y_0))
\end{aligned}
\tag{1}
$$

Successive counter values are generated using the function incr $\;)$, which treats the rightmost 32 bits of its argument as a nonnegative integer with the least significant bit on the right, and increments this value modulo 2^{32}. More formally, the value of incr $\; F \| I)$ is $F \| \;\; I + 1 \bmod 2^{32})$. The encryption process is illustrated in Figure 1.

The function GH SH is defined by GH SH $\; H, A, C) = X_{m+n+1}$, where the inputs A and C are formatted as described above, and the variables X_i for $i = 0, \ldots, m+n+1$ are defined as

$$
X_i = \begin{cases}
0 & \text{for } i = 0 \\
\;\; X_{i \quad 1} \oplus A_i) \cdot H & \text{for } i = 1, \ldots, m \quad 1 \\
\;\; X_{m \quad 1} \oplus \;\; A_m^* \| 0^{128 \quad v})) \cdot H & \text{for } i = m \\
\;\; X_{i \quad 1} \oplus C_{i \quad m}) \cdot H & \text{for } i = m+1, \ldots, m+n \quad 1 \\
\;\; X_{m+n \quad 1} \oplus \;\; C_n^* \| 0^{128 \quad u})) \cdot H & \text{for } i = m+n \\
\;\; X_{m+n} \oplus \;\; \text{len} \;\; A) \| \text{len} \;\; C))) \cdot H & \text{for } i = m+n+1.
\end{cases}
\tag{2}
$$

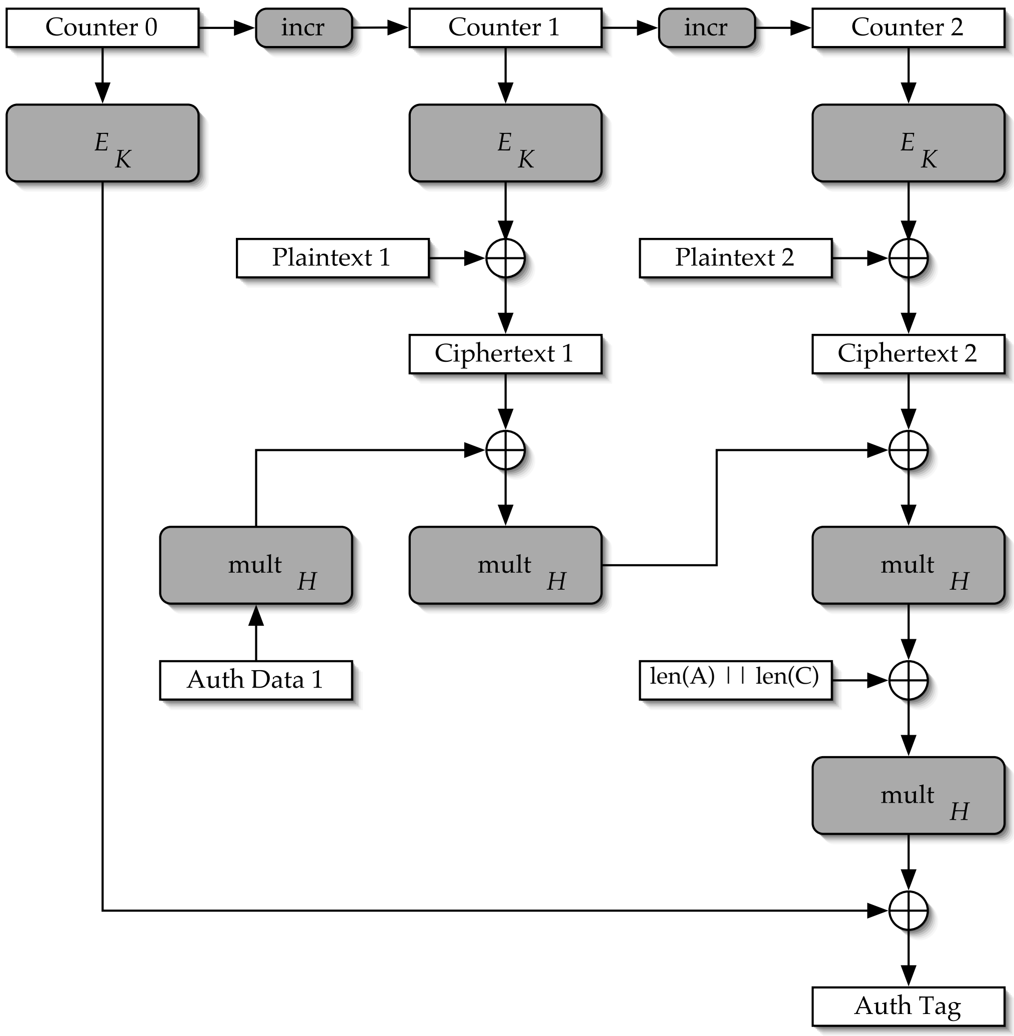

Figure 1: The authenticated encryption operation. For simplicity, a case with only a single block of additional authenticated data (labeled uth Data 1) and two blocks of plaintext is shown. Here E_K denotes the block cipher encryption using the key K, mult denotes multiplication in $GF\ 2^{128}$) by the hash key H, and incr denotes the counter increment function.

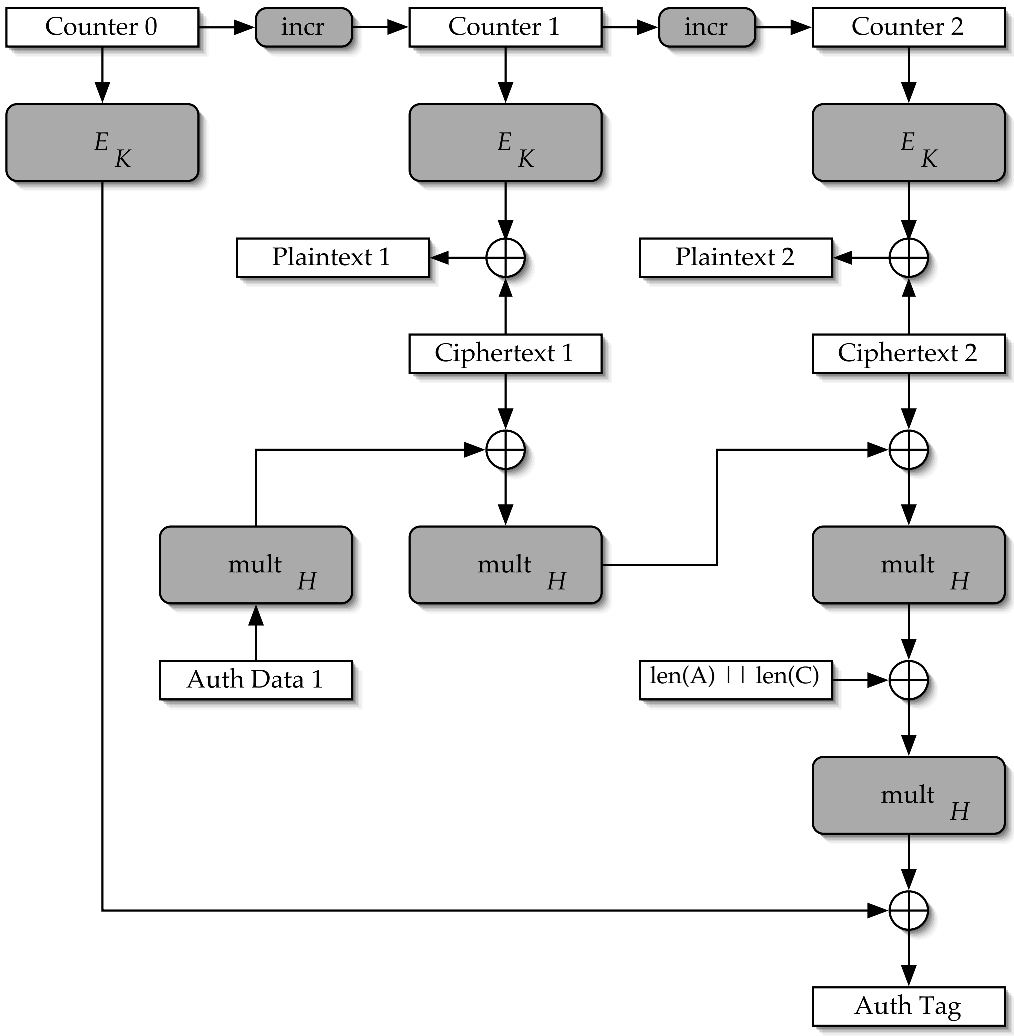

Figure 2: The authenticated decryption operation, showing the same case as in Figure 1.

2.4 Decryption

The authenticated decryption operation is similar to the encrypt operation, but with the order of the hash step and encrypt step reversed. More formally, it is defined by the following equations:

$$
\begin{aligned}
H &= E\ \ K, 0^{128}) \\
Y_0 &= \begin{cases} IV \| 0^{31}1 & \text{if len } \ IV) = 96 \\ \text{GH SH } H, \{\}, IV) & \text{otherwise.} \end{cases} \\
T' &= \text{MSB}_t \ \text{GH SH } H, A, C) \oplus E\ \ K, Y_0)) \\
Y_i &= \text{incr } \ Y_i \ \ {}_1) \text{ for } i = 1, \ldots, n \\
P_i &= C_i \oplus E\ \ K, Y_i) \text{ for } i = 1, \ldots, n \\
P_n^* &= C_n^* \oplus \text{MSB}_u \ E\ \ K, Y_n))
\end{aligned}
$$

The tag T' that is computed by the decryption operation is compared to the tag T associated with the ciphertext C. If the two tags match (in both length and value), then the ciphertext is returned. Otherwise, the special symbol **F IL** is returned. The decryption process is illustrated in Figure 2.

2.5 Multiplication in $G \quad 2^{128})$

The multiplication operation is defined as an operation on bit vectors in order to simplify the specification. This definition corresponds to the particular choice of the field representation used in GCM. Section 3 provides background information on this field and its representation, and Section 4 describes some strategies for efficient implementation.

Each element is a vector of 128 bits. The i^{th} bit of an element X is denoted as X_i. The leftmost bit is X_0, and the rightmost bit is X_{127}. The multiplication operation uses the special element $R = 11100001 \| 0^{120}$, and is defined in lgorithm 1. The function rightshift) moves the bits of its argument one bit to the right. More formally, whenever $W =$ rightshift $V)$, then $W_i = V_i \ \ {}_1$ for $1 \le i \le 127$ and $W_0 = 0$.

3 The Field $G \quad 2^{128})$

finite field is defined by its multiplication and addition operations. These operations obey the basic algebraic properties that one expects from multiplication and addition (commutativity, associativity, and distributivity). Both operations map a pair of field elements onto another field element. In a polynomial basis, the multiplication of two elements X and Y consists of multiplying

lgorithm 1 Multiplication in $GF\ 2^{128})$. Computes the value of $Z = X \cdot Y$, where X, Y and $Z \in GF\ 2^{128})$.

$Z \leftarrow 0, V \leftarrow X$
for $i = 0$ to 127 **do**
  if $Y_i = 1$ **then**
    $Z \leftarrow Z \oplus V$
  end if
  if $V_{127} = 0$ **then**
    $V \leftarrow$ rightshift $V)$
  else
    $V \leftarrow$ rightshift $V) \oplus R$
  end if
end for
return Z

the polynomial representing X with the polynomial representing Y, then dividing the resulting 256-bit polynomial by the field polynomial; the 128-bit remainder is the result. We describe this operation in more detail below. The field polynomial is fixed and determines the representation of the field. GCM uses the polynomial $f = 1 + \quad + \quad ^2 + \quad ^7 + \quad ^{128}$.

The addition of two elements X and Y consists of adding the polynomials together. Because each coefficient is added independently, and the coefficients are in $GF\ 2)$, this operation is identical to the bitwise exclusive-or of X and Y. No reduction operation is needed. Subtraction over $GF\ 2^{128})$ is identical to addition, because the field $GF\ 2)$ has that property.

To describe multiplication, we take the small first step of showing how to multiply a field element X by the field element P defined by

$$P_i = \begin{matrix} 1 & \text{for } i = 1 \\ 0 & \text{otherwise,} \end{matrix} \tag{3}$$

then we show how to use this method to multiply any two field elements. The element P corresponds to the polynomial . The multiplication of a polynomial by is simple; it corresponds to a shift of indices:

$$\cdot \ x_0 + x_1 \ ^1 + x_2 \ ^2 + \ldots + +x_{127} \ ^{127}) = x_0 \ + x_1 \ ^2 + x_2 \ ^3 + \ldots + +x_{127} \ ^{128}. \tag{4}$$

If $x_{127} = 0$, then the product is a polynomial of degree 127. Otherwise, we must divide the result by the field polynomial f to find the remainder. To find the remainder of a polynomial $^{128} + a$, where $a = a_0 + a_1 \ + a_2 \ ^2 + \ldots + a_{127} \ ^{127}$ is a polynomial of degree 127, we need to find polynomials q and r such that $^{128} + a = q \cdot f + r$, where the remainder r has degree 127. We can solve this equation for r when $q = 1$:

$$r = \ ^{128} + a \quad f = a + 1 + \quad + \ ^2 + \ ^7. \tag{5}$$

The highest term of f is canceled away (since addition is over $GF\ 2)$), and the net effect is just to add the lowest terms of f to a. To compute $Y = X \cdot P$, we combine the two steps of shifting coefficients and adding in the lowest terms of f if the highest term of X is equal to one. In bit operations, this can be expressed as

if $X_{127} = 0$ **then**
  $Y \leftarrow$ rightshift $X)$
else
  $Y \leftarrow$ rightshift $X) \oplus R$
end if

where R is the element whose leftmost eight bits are 11100001, and whose rightmost 120 bits are all zeros.

In order to multiply two arbitrary field elements X and Y, we can express Y in terms of P, then use the method described above. This method is used in the following algorithm, which takes X and Y as inputs and returns their product.

$Z \leftarrow 0, V \leftarrow X$
for $i = 0$ to 127 **do**
  if $Y_i = 1$ **then**
    $Z \leftarrow Z \oplus V$
  end if
  $V \leftarrow V \cdot P$
end for
return Z

In this algorithm, V runs through the values of $X, X \cdot P, X \cdot P^2, \ldots$, and the powers of P correspond to the powers of , modulo the field polynomial f. This method is identical to lgorithm 1, but is defined in terms of field elements instead of bit operations.

4 Implementation

Implementing GCM is straightforward in both hardware and software given an implementation of the underlying block cipher and the multiplication operation over $GF\ 2^{128})$. In this section, we provide an overview of efficient hardware and software implementations, and a detailed description of the multiplication operation in software.

The number of block cipher invocations needed to encrypt an p-bit plaintext with ES GCM is equal to $\lceil p/128 \rceil + 1$. The same number of multiplications over $GF\ 2^{128})$ are needed. n additional block cipher invocation is needed to compute the hash key H if it is not stored. If there are an additional q bits of data to be authenticated, then an additional $\lceil q/128 \rceil$ multiplications are needed. The decrypt operation is similar to the encrypt operation and shares its performance characteristics. We provide more detailed performance data for the implementation methods discussed below.

4.1 Software

Multiplication in a binary field can use a variety of time-memory tradeoffs. It can be implemented with no key-dependent memory, in which case it will generally run several times slower than ES. Implementations that are willing to sacrifice modest amounts of memory can easily realize speeds greater than that of ES.

The operation $H \cdot X$ is linear in the bits of X, over the field $GF\ 2)$. This property can be exploited to make efficient table-driven implementations, in which tables computed for a particular value of H can be used to multiply H by an arbitrary element X. The simplest method computes $Z = X \cdot H$ as

$$Z = M_0[\text{byte } X, 0)] \oplus M_1[\text{byte } X, 1)] \oplus \ldots \oplus M_{15}[\text{byte } X, 15)], \tag{6}$$

where byte $X, i)$ denotes the i^{th} byte of the element X.

To show how this method works, we introduce a decomposition of a field element into a sum of field elements which have only eight nonzero bits. We denote as $\mathcal{S}$ the set of elements in $GF\ 2^{128})$ that have the rightmost 120 bits equal to zero. For any element $A \in \mathcal{S}$, multiplying A with any other element $H \in GF\ 2^{128})$ is relatively simple. For a fixed element H we can construct a table M such that the product $A \cdot H$ can be computed easily. Because there are only $2^8 = 256$ elements in $\mathcal{S}$, the table can be constructed by looping over all 256 cases and computing each result.

Our decomposition of X is described by the equation

$$X = \bigoplus_{i=0}^{15} x_i \cdot P^{8i}, \text{ where } x_i \in \mathcal{S} \text{ for all } i, \tag{7}$$

where P is the element associated with and defined in Equation 3. In this decomposition, the element x_i has the i^{th} byte of X as its nonzero part. The product $H \cdot X$ can be expressed as

$$H \cdot X = \bigoplus_{i=0}^{15} x_i \cdot H \cdot P^{8i} = \bigoplus_{i=0}^{15} M_i[\text{byte } X, i)]. \tag{8}$$

Each table $M_0, M_1, \ldots M_{15}$ is initialized before the multiplication algorithm is run so that its entries satisfy the equation $M_i[\text{byte } X, i)] = x_i \cdot H \cdot P^{8i}$. Each of the 16 tables holds 256 values, each of which is 16 bytes long, for a total of 65,536 bytes. When this table is used in GGM, it is key-dependent and must be computed at key initialization time and stored along with the key. To conserve memory, we could instead decompose X into 32 components with four bits each, for a total of 8,192 bytes.

With a small increase in the amount of computation, we can reduce the storage requirements considerably, as described by Shoup [9]. We can use only the table M_0 defined above to multiply an arbitrary element $X \in GF\ 2^{128})$ by H as follows. We first express the product as

$$H \cdot X = H \cdot \bigoplus_{i=0}^{15} x_i \cdot P^{8i} = \bigoplus_{i=0}^{15}\ x_i \cdot H) \cdot P^{8i},$$

This equation leads to the following simple algorithm:

$Z \leftarrow 0$
for $i = 15$ to 1 **do**
  $Z \leftarrow Z \oplus\ x_i \cdot H)$
  $Z \leftarrow Z \cdot P^8$
end for
$Z \leftarrow Z \oplus\ x_0 \cdot H)$
return Z

Note that i loops from 15 down to zero so that the rightmost byte is associated with the lowest power of P^8. In order to use this method, we need an efficient way to compute $X \cdot P^8$ for an arbitrary element X. We make use of the decomposition in Equation 7 again, and express the product as

$$X \cdot P^8 = \bigoplus_{i=0}^{15} x_i \cdot P^{8\ i+1)}. \tag{9}$$

The expression $x \cdot P^{8\ i+1)}$, for $x \in \mathcal{S}$ and $0 \leq i < 15$, corresponds to a simple bit-rotation of the element x to the right by eight bits. Thus the first 15 terms on the right-hand side of Equation 9 can be computed with only a rotation. The expression $x \cdot P^{128}$ is not so simple, but can be computed using a table, as we did above. In the following, we assume that there is a table R containing these products, so that $R[\text{byte}\ x, 0)] = x \cdot P^{128}$ for all $x \in \mathcal{S}$. lgorithm 2 details how these methods can be combined.

The table M_0 requires only 4096 bytes of storage. Each elements of the table R has its rightmost 112 bits equal to zero. In practice, those bits need not be stored, so that table contains 1024 bytes. It is not key-dependent. Storage requirements can be reduced further by using a decomposition into four-bit elements, so that M_0 and R consume 256 bytes and 64 bytes, respectively.

The performance of these methods is outlined in Table 1, which gives the throughput for a C implementation of GH SH using the strategies discussed above on a Motorola G4 processor (a 32-

lgorithm 2 Computes $Z = X \cdot H$ using the tables M_0 and R.

$Z \leftarrow 0$
for $i = 15$ to 1 **do**
  $Z \leftarrow Z \oplus M_0$[byte $X, i)$]
  $A \leftarrow$ byte $X, 15)$
  for $j = 15$ to 1 **do**
    byte $Z, j) \leftarrow$ byte $Z, j \quad 1)$
  end for
  $Z \leftarrow Z \oplus R[A]$
end for
$Z \leftarrow Z \oplus M_0$[byte $X, 0)$]
return Z

Method	Storage requirement	Throughput (cycles per byte)
Simple, 8-bit tables	65,535 bytes/key	13.1
Simple, 4-bit tables	8,192 bytes/key	17.3
Shoup's, 8-bit tables	1024 bytes + 4096 bytes/key	32.1
Shoup's, 4-bit tables	64 bytes + 256 bytes/key	69.3
No tables	16 bytes/key	119

Table 1: The throughput of GH SH using various different methods for the Galois field multiplication on a Motorola G4 processor.

bit RISC CPU). These times should be compared to that of the OpenSSL [10] optimized C version of ES, which ran at 33.0 cycles per byte on the same platform, and requires 4096 bytes + 160 bytes/key of storage. The GNU C compiler (gcc version 3.3) was used in all cases.

Because the computation of the tables used in multiplication introduces a delay between the time that a key is provided and the time that the key can be used, it is desirable to minimize amount of time required for that computation. In lgorithm 3 we outline a method for computing table M_0 quickly. The efficiency in this method comes from the use of the information contained in the parts of the table that have already been computed to find the remaining entries. Each table entry that has an index that is a power of two contains a product of H times a power of P. The other elements of the table can computed by summing together these elements. For example, $M_0[128] = H$ and $M_0[64] = H \cdot P$. The index 192 (decimal) has a binary decomposition of 11000000, so $M_0[192] = H \oplus H \cdot P = M_0[64] \oplus M_0[128]$. The table can be computed using lgorithm 3, which makes a single pass over the data, using only 247 field additions and eight 'mutliply by P' operations. The other tables $M_1, M_2, \ldots, M_{15}$ can be computed using similar algorithms.

lgorithm 3 Computes the table M_0 given an element $H \in GF\ \ 2^{128}$).

$M[128] \leftarrow H$, $i \leftarrow 64$
while $i > 0$ **do**
  $M[i] \leftarrow M[2i] \cdot P$
  $i \leftarrow \lfloor i/2 \rfloor$
end while
$i \leftarrow 2$
while $i < 128$ **do**
  for $j = 1$ to $i \quad 1$ **do**
    $M[i + j] = M[i] \oplus M[j]$
  end for
  $i \leftarrow 2i$
end while
$M[0] \leftarrow 0^{128}$
return M

4.2 Hardware

In this section, we outline a pipelined hardware design, which is illustrated in Figure 3. The trapezoids at the top and bottom denote inputs and outputs, respectively. The rhomboids denote the points at which data paths are switched. There are three inputs: data that is authenticated-only (D), the IV, and the plaintext. The IV is fed into the increment function, which then outputs successive counter values that are fed into the block cipher pipeline, shown as E_K in the figure. The first encrypted counter is sent to encrypt the GH SH output (path 3), then the output of that function is switched so that the other encrypted counters are exored with the plaintext to form the ciphertext (path 2). The authenticated-only data is fed into the GH SH function (path 1), then the input of that function is switched to the ciphertext (path 2). fter all of the data input to GH SH has been processed, the output of that function is exored with the fist encrypted counter, producing the authentication tag. In this design, the tag-generating pipeline and ciphertext-generating pipelines are independent, except for the tag-encryption step. These two pipelines can be made completely independent by adding another ES engine dedicated to the encryption of the GH SH output.

Binary Galois field multiplication is especially suitable for hardware implementations. Many implementation strategies are discussed in the literature. Parr [11] summarizes the efficiency of various finite field multiplication methods for $GF\ \ 2^q$) as follows:

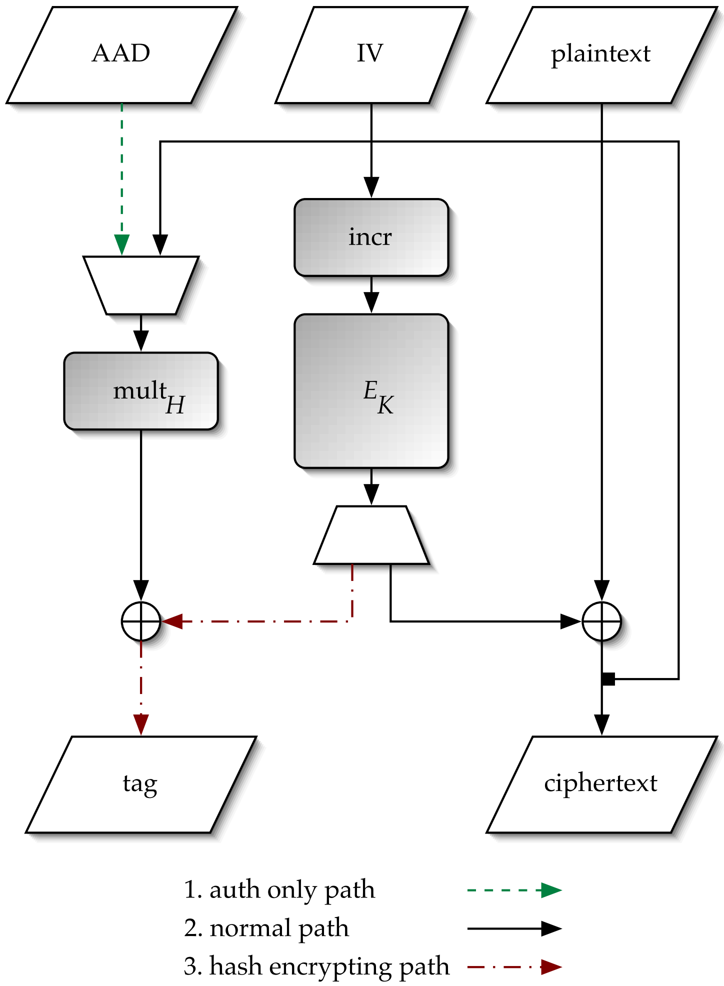

Figure 3: hardware implementation of GCM, showing the different data paths through the circuit.

Method	Time	rea
Parallel	1	$\mathcal{O}\ q^2)$
Digit Serial [12]	q/D	$\mathcal{O}\ qD)$
Bit Serial	q	$\mathcal{O}\ q)$
Super Serial [13]	qE	$\mathcal{O}\ q/E)$

The bit serial method is a direct implementation of lgorithm 1. The parallel method computes the product in a single clock; it essentially unwinds the q loops of the bit serial method. The other methods trade off circuit area against computation time. With $q = 128$, the parallel method is practical, and it can keep up with any pipelined implementation of ES. In many cases, the digit serial method may provide a worthwhile tradeoff; it has performance parameters between the serial and parallel methods. lgorithm 2 is structurally similar to a digit serial circuit, though a hardware design would use a dedicated circuit rather than a table.

The multiply operation can be performed in a single clock, or a small number of clocks, without its area cost dominating the total cost of the GCM circuit. Thus a straightforward implementation using a single digit serial or parallel multiplier appears to be useful. lternately, it is possible to parallelize the multiplication step, as observed in [6]. For example, there can be two multipliers, one of which works on the even blocks $X_0, X_2, X_4, \ldots$, and one of which works on the odd blocks $X_1, X_3, X_5, \ldots$ in Equation 2. This method follows from the fact that

$$X_5 \cdot K \oplus X_4) \cdot K \oplus X_3) \cdot K \oplus X_2) \cdot K \oplus X_1) \cdot K \oplus X_0) \cdot K =$$
$$X_5 \cdot K^2 \oplus X_3) \cdot K^2 \oplus X_1) \cdot K^2\big] \oplus \quad X_4 \cdot K^2 \oplus X_2) \cdot K^2 \oplus X_0) \cdot K\big] \,.$$

5 Using GCM

This section illustrates some uses of GCM. n example use for protecting a network packet flow is shown in Figures 4 and 5, which include a typical cryptographic encapsulation, modeled after the IEEE 802 Media ccess Control Security draft standard [14]. data field is encrypted and authenticated, and is carried along with a header and a sequence number. The header is authenticated by including it in the D. The sequence number is included in the IV. The authentication tag is carried in an Integrity Check Value (ICV) field. Note that there is no need to pad the plaintext, since any length can be provided as an input. In the authentication decryption operation (Figure 5), these fields provide the inputs. The plaintext is the output, unless the authentication check failed. In that case, the decrypt operation would return **F IL** rather than the plaintext, and the decapsulation would halt and the plaintext would be discarded rather than forwarded or further processed. fter the operation, the header and sequence number can be checked, and their values can be trusted.

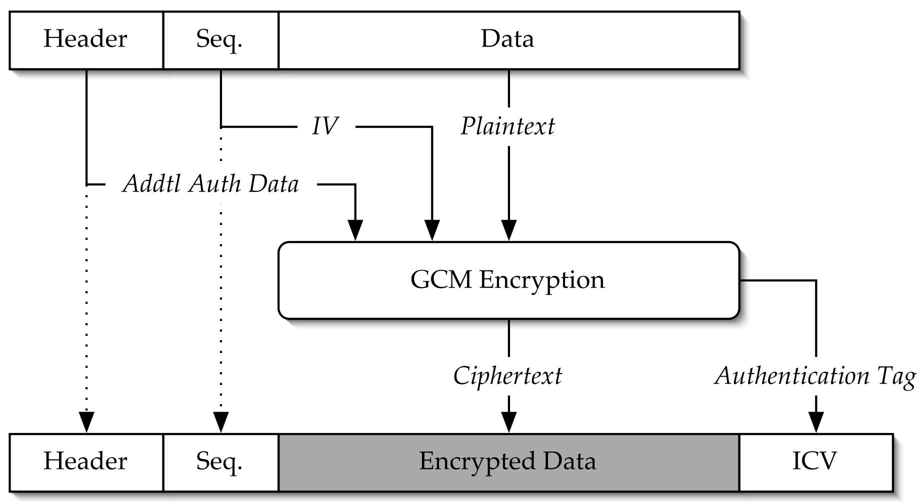

Figure 4: Using GCM to encrypt and authenticate a packet, showing how the fields of the security encapsulation map onto the inputs and outputs of the authenticated encryption mode.

By including the sequence number in the IV, we can satisfy the requirement that IV values be unique. If that number is less than 96 bits long, it can be concatenated with another value in order to form the IV. This other value could be constant, such as a string of zeros, or it could be a random string, which adds to the security of the system because it makes the inputs less predictable than they would be otherwise. The data needed to form the IV has to be known to both the encrypt side and the decrypt side, but it need not all be included in the packet.

In some situations, it may be desirable to have the same GCM key used for encryption by more than one device. In this case, coordination is needed to ensure the uniqueness of the IV values. simple way in which this requirement can be met is to include a device-specific value in the IV, such as a network address.

6 Properties and Rationale

The important properties of GCM are summarized in Table 2. Its primary motivation is the need for an authenticated encryption mode that can be efficiently implemented in hardware at very high data rates, achieves high performance in software, is provably secure, and is free of intellectual property restrictions. These goals are important for high-speed network security, especially at the link layer. This point is underscored by the fact that the IEEE 802.1 M C Security Task Group has proposed to use mode as the standard's mandatory-to-implement cryptoalgorithm [14].

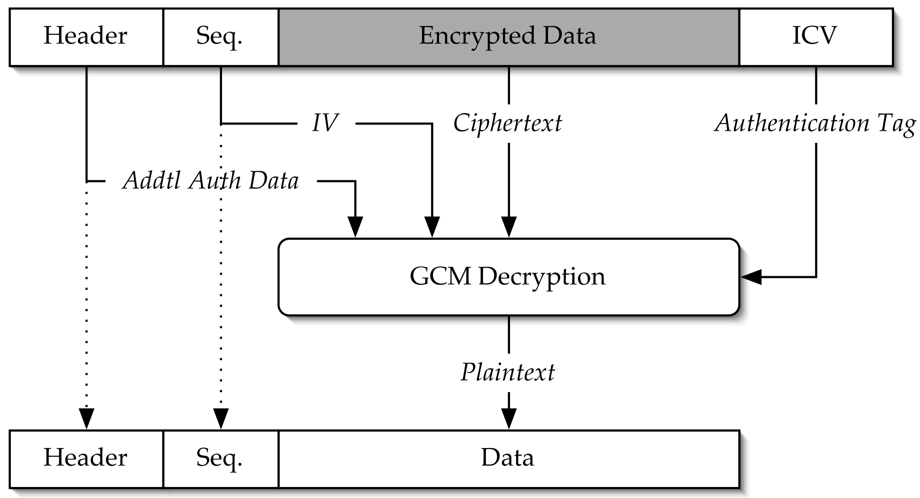

Figure 5: Using GCM to decrypt and verify the authenticity of a packet.

Security Function	uthenticated encryption
Error Propagation	None
Synchronization	Same IV used by sender and recipient
Parallelizability	Encryption - block-level uthentication - bit-level
Keying Material Requirements	One block cipher key
IV Requirements	Each IV must be distinct, for each fixed key IV can have arbitrary length from 1 to 2^{64} bits
Memory Requirements	Same as block cipher
Pre-processing capability	Keystream can be precomputed Fixed parts of IV or A can be processed in advance Effective methods available for accelerating authentication; recommended methods (Section 4.1) use 256b to 64 Kb
Message Length Requirements	rbitrary message up to 2^{39} 256 bits rbitrary additional authenticated data up to 2^{64} bits No padding
Ciphertext Expansion	Ciphertext length is identical to plaintext length 0 to 128 bits required for the authentication tag
Other Features	Can be used as a stand-alone M C Can be used as an incremental M C On-line (message lengths need not be known in advance) Minimal circuit depth

Table 2: summary of the properties of GCM.

Defining a mode using generic composition (encrypt-then-authenticate [15]) is a simple way to achieve a provably secure mode of operation, as long as the underlying components for encryption and authentication are provably secure. Our strategy, similar to that of CWC and E X, was to start with generic composition and then modify the algorithm in provably secure ways that better address other requirements.

Counter mode is the obvious choice for the foundation of any authenticated encryption mode, since it is the one well-known encryption-only mode that is fully parallelizable. Our choice of M C represents the best available solution between hardware efficiency (particularly, parallelizability and memory requirements), software efficiency, security bound and intellectual property restrictions. Like CWC, we chose a parallelizable M C based on the Carter-Wegman design [16] that uses polynomial hashing. However, we used a binary Galois field rather than a 127-bit integer field because of the ease and efficiency with which binary fields can be implemented in hardware. Because our primary motivation is to achieve high data rates, the choice of a field with hardware-friendly multiplication operations is natural. Perhaps surprisingly, the binary field makes it easier to realize high-speed software implementations, as shown below.

In hardware, GCM adds a negligible amount of overhead compared to a pipelined ES implementation. OCB would share similar properties, except that it requires both an ES encryption and ES decryption engine. CWC, on the other hand, has an expensive message authentication function. While it is capable of high speeds, the implementation is significantly more expensive than an ES encryption unit. typical fully pipelined implementation of a single ES counter mode encryption engine requires approximately 90K gates. In the same environment a straightforward implementation of GCM requires only an additional 30K gates for the binary field hash function, by our estimation. In contrast, CWC mode requires the same number of gates for encryption, but requires over 100K gates for its integer-based hash function [6].

We worked hard to ensure that GCM would never unnecessarily stall data pipelines. In a high-speed network implementation, it is essential to minimize the circuit depth in order to preserve performance on short packets or frames. pipelined implementation of ES will have a latency of about ten clock cycles, since each round can be computed in a single clock (this latency can be reduced, but only with significant cost). typical high-speed cipher implementation processes 128 bits per clock cycle. t a clock rate of 200 megahertz, it runs at 25 gigabits per second. Stalling the pipeline for ten clock cycles would consume as much time as is required to process 1280 bits of data - resulting in a 50% performance degradation for 160 octet packets. The circuit depth of GCM is essentially that of the block cipher, assuming that the hash key has been computed and stored. In contrast, the CWC mode of operation has a circuit depth that is longer by an additional application of the block cipher, due to an additional post-processing step [6, Step 1 of Section 2.5]. This additional delay significantly impacts its performance on short packets or frames. Unfortunately, this problem cannot be fixed with a simple change to CWC. It needs the post-hashing block cipher invocation to allow the authentication tag to have a size less than the block width of the cipher, because the integer field does not have the same properties as does the binary field.

The field definition was chosen as follows. The field polynomial was chosen to have low weight and terms of low order, properties that promote efficient implementations; the polynomial was referenced from the table of low weight binary irreducible polynomials of Seroussi [17]. We chose to use a 'little endian' definition of the field, in which multiplication proceeds from left to right. This property allows a multiplier to process data as it arrives, using the algorithms described above, whenever the width of the data bus is less than 128 bits.

We allow the IV to have an arbitrary length, but we include an important optimization that ensures that pipeline stalls are avoided when the length of the IV is 96 bits. Otherwise the encryption pipeline could stall until the initial counter value Y_0 is computed. However, even in environments where a larger IV is used, it is possible to take advantage of precomputation to minimize or eliminate pipeline stalls. We believe in giving the user an option to have an arbitrary-sized IV, because it eases the job of using the mode. When application and protocol designers specify an IV format for our mode, they don't need to adopt the procrustean approach of fitting their data into a small, fixed number of bits. Importantly, GCM is secure even if IVs of different length are used with the same key, as long all IV values of the same length are distinct.

lso helping to avoid pipeline stalls is the fact that GCM is *on-line*, meaning that it does not need to know the length of a message in advance. Instead, it can calculate the length of the message as it arrives. CCM does not have this property, which can be a particular problem with large messages, as hardware implementations may find it expensive to provide the memory necessary to buffer the largest feasible messages.

GCM uses only the forward (encrypt) direction of its block cipher for both encryption and decryption. This fact simplifies the implementation of ES GCM. In software, it is not necessary to include the code and tables that are needed for ES decryption, and in hardware, there is no need to design or include an ES decryption circuit. This property is shared by the CWC, CCM, and E X modes, but not OCB, which uses both the forward and backward directions of its cipher.

The choice of a 128-bit field allows message authentication with that level of security. The use of a smaller field would have resulted in a modest reduction in the computational cost of the multiplication operation. However, we found that the use of field elements that match the ES block size simplifies implementations considerably, avoiding complex byte-shifting in software and potential data buffering issues at high speed. To have used a smaller field would have added complexity without much gain in performance, and would have reduced the security level. This design choice reflects knowledge gained by implementation experience with CWC, the hash function of which operates on 96-bit blocks.

GCM is also suitable for software implementations. It is simple to build a portable implementation of GH SH that outperforms ES on all platforms, with a modest amount of key-dependent

Mode	**Message size (bytes)**				
	16	64	256	1024	8192
CBC-HM C-SH 1	1270	342	124	68.4	51.2
CCM	159	75.6	54.5	49.2	47.6
CWC	227	102	72.7	63.3	61.2
E X	239	93.8	59.4	51.1	48.0
GCM, 64Kb storage	60.8	44.8	36.1	36.6	38.1
GCM, 8Kb storage	89.9	51.9	42.9	43.0	40.1
GCM, 4Kb storage	118	69.1	46.5	54.1	53.5
GCM, 256b storage	179	108	89.5	85.4	84.6
OCB	89.4	43.3	31.4	29.3	29.0

Table 3: Software performance for various different ES modes of operation, expressed in clock cycles per byte.

precomputation. In contrast, the efficiency of CWC implementations varies depending on the performance and characteristics of the underlying multiplication operation. It usually requires a substantial effort to get a software-based CWC implementation to run as quickly as ES on 32-bit platforms. On 16-bit and 8-bit platforms, CWC's performance drops to unacceptable levels, requiring a minimum of 48 multiplies and nearly as many additions on a 16-bit platform, and an unacceptable 192 multiplies on an 8-bit platform. CWC's dependence on multiplication does lead to it being a bit faster than GCM on 64-bit platforms. well-optimized OCB implementation will always be faster than CWC, and will generally be faster than GCM, with the exception of small messages (see Table 3). The software speed advantage of OCB is slight, and the speed of both modes are dominated by that of the underlying block cipher. This relative advantage for OCB is probably outweighed by GCM's other advantages, including its lack of intellectual property restrictions.

The software performance for various modes of operation is captured in Table 3. We started with reference versions of each algorithm and then modified them to all use the same underlying ES implementation. The GCM implementations were our own, the OCB implementation was written by Ted Krovitz and the remainder of the implementations were written by Brian Gladman. ll of the performance times were found experimentally on a G4 processor, using the GNU C compiler version 3.3 with the options `O3 funroll loops ftracer fnew ra`[2]. For comparison, we include a generic composition of CBC with HM C-SH 1, which is a common method for achieving authenticated encryption. Each of the implementation methods for GCM discussed in Section 4.1 is included, labeled by the amount of state that it requires. OCB is the fastest, except on messages of 64 bytes or smaller, where GCM with 64K storage has the same or greater speed. For large messages, the simple implementation strategy for GCM never runs in more than 4/3 the

[2]The option `funroll loops` was not used for CWC, since it degraded the performance of that algorithm. Interestingly, the new compiler options `ftracer` and `fnew ra` improved ES performance considerably, raising the relative performance of CCM and E X, which call ES twice per message block.

time of OCB, even when storage is limited to 8K. EAX, CCM, and CBC-HMAC-SHA1 are slower, especially on shorter messages. GCM with 4K storage is faster than those modes on messages less than 1024 bytes in length, and GCM with 256 byte storage is faster than CBC-HMAC-SHA1 on the same messages. For storage comparison, the performance-optimized implementation of the generic composition requires 180 bytes of storage per key (for the AES expanded key and a single SHA1 context). These results clearly establish the viability of GCM for high-speed software implementations.

Only a single key is input into GCM. The hash key is derived from this key, and is used both for message authentication and for IV-processing. By using the same key for both purposes, we reduce the amount of storage needed. The use of a single key simplifies the interface to the mode, and also reduces the storage requirement by allowing implementations to store only the block cipher key and derive the hash key from it during the encryption or decryption operations.

GCM can be used as a stand-alone message authentication code, if authentication but not encryption is needed, simply by having the plaintext P be zero-length. OCB, which cannot accept additional authenticated data whose size exceeds the block length of the cipher, cannot be used in this way. GCM also has the useful property that it can be used as an incremental MAC [7]. Such constructions can be used in applications in which a large and dynamic data set must be authenticated such as a remote database, file system, or network storage. In such situations, a conventional MAC is often unworkable. The only other incremental MAC [18] of which the authors are aware is the subject of patent claims. None of the other proposed modes have this property.

Our mode inherits several desirable properties from counter mode. The ciphertext has minimal expansion; it will be exactly the same length as the plaintext. The only expansion in message size comes from the authentication tag. The IV need not be random, as it must be with CBC mode; a sequential IV value is sufficient. This weaker requirement is easier to satisfy, since randomness is often a precious resource in a cryptomodule. We are aware of more than one CBC implementation whose security suffered from a poor choice of IV selection.

Counter mode can be implemented in many different ways; we have followed current practice in order to simplify adoption. The counter format and increment function that are used matches that in the proposed IPsec ESP Counter Mode standard [19]. This format avoids the need to implement a 128-bit integer increment in hardware (which has a high circuit complexity at high speeds) or an LFSR in software (which would take about eight times as many clock cycles on a 32-bit CPU).

Software implementations of network security protocols often favor cryptographic encapsulations that allow the receiver of a bogus message to discard it before encryption, in order to avoid that step. GCM has that property, since the ciphertext, not the plaintext, is authenticated.

7 Security

GCM is secure in the concrete security models introduced by Bellare, Killian, and Rogaway [20] for message authentication, and Bellare, Desai, Jokipii, and Rogaway for confidentiality [22], against adversaries that can adaptively choose the plaintext, the additional associated data, and the IV (as long as the requirements on these inputs are respected). Its security relies on the fact that the underlying block cipher cannot be distinguished from a random permutation, an assumption which is common in cryptographic designs and which appears to be valid for the ES.

The security of its basic components are well established. The use of universal hashing for provably strong message authentication was introduced by Carter and Wegman in 1981 [16], and that method has been an element in the design of many cryptosystems since that time. Counter mode was suggested in 1979 by Diffie and Hellman [21], and was shown to be secure in a strong, concrete sense by Bellare et. al. [22]. While the proof of security for GCM rests on those proofs, there are some differences. The derivation of the hash key H from the block cipher key K, the hashing of the IV, and the use of that key for both IV-processing and message authentication are important details. More information is available in a separate security analysis [23].

References

[1] DES Modes of Operation. *Federal Information Processing Standards Publication 81*, December, 1980.

[2] D. Whiting, N. Ferguson, and R. Housley. Counter with CBC-M C (CCM). *Submission to NIST*, 2002. vailable online at `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/`.

[3] M. Bellare, P. Rogaway, and D. Wagner, conventional authenticated-encryption mode. 2003. vailable online at `http://eprint.iacr.org/2003/069/`.

[4] T. Iwata, K. Kurosawa. OM C: One-Key CBC. vailable online at `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/`.

[5] P. Rogaway, M. Bellare, J. Black, and T. Krovitz. OCB: a block-cipher mode of operation for efficient authenticated encryption. *CM CCS*, 2001.

[6] T. Kohno, J. Viega, and D. Whiting. The CWC- ES Dual-use Mode. *Internet Draft, Crypto Forum Research Group*, May 20, 2003. Work in progress. vailable online at `http://www.zork.org/cwc/draft irtf cfrg cwc 01.txt`.

[7] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: the case of hashing and signing. *dvances in Cryptology - Proceedings of CRYPTO '94*.

[8] M. Dworkin, Recommendation for Block Cipher Modes of Operation: Methods and Techniques, *NIST Special Publication 800-38* .

[9] V. Shoup, On Fast and Provably Secure Message uthentication Based on Universal Hashing, *dvances in Cryptology - Proceedings of CRYPTO '96*, 1996.

[10] Multiple authors. OpenSSL: The Open Source toolkit for SSL/TLS. `www.openssl.org`.

[11] C. Parr, Implementation Options for Finite Field rithmetic for Elliptic Curve Cryptosystems. *ECC '99*.

[12] L. Song, K.K. Parhi , Efficient Finite Field Serial/Parallel Multiplication. *1996 International Conference on pplication-Specific Systems, rchitectures, and Processors (S P '96)*, ugust, 1996.

[13] G.Orlando and C.Paar. super-serial Galois field multiplier for FPG s and its application to public key algorithms. *Proceedings of the IEEE Symposium on Field-programmable custom computing machines (FCCM '99)*, pages 232-239, 1999.

[14] . Romanow, Ed. Media ccess Control (M C) Security. *IEEE 802.1 E*, Draft Standard. Work in progress.

[15] H. Krawczyk. The Order of Encryption and uthentication for Protecting Communications. *In Lecture Notes in Computer Science*, volume 2139, 2001.

[16] M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265279, 1981.

[17] G. Seroussi. Table of Low-Weight Binary Irreducible Polynomials. *HP Labs Technical Report HPL-98-135*, Computer Systems Laboratory, ugust, 1998.

[18] M. Bellare, R. Guerin, and P. Rogaway. XOR M Cs: New methods for message authentication using finite pseudorandom functions. *dvances in Cryptology - Proceedings of CRYPTO '95*.

[19] R. Housley, Using ES Counter Mode With IPsec ESP. *IETF Internet Draft* `draft ietf ipsec ciph aes ctr 05.txt`. Work in Progress.

[20] M. Bellare, J. Killian, and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, Vol. 61, No. 3, Dec 2000, pp. 362–399.

[21] W. Diffie and M. Hellman. Privacy and uthentication: n Introduction to Cryptography. *Proceedings of the IEEE*, Volume 67, Number 3, March, 1979.

[22] M. Bellare, . Desai, E. Jokipii and P. Rogaway. Concrete Security Treatment of Symmetric Encryption: nalysis of the DES Modes of Operation. *Proceedings of 38th nnual Symposium on Foundations of Computer Science, IEEE*, 1997.

[23] D. McGrew and J. Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. *Proceedings of INDOCRYPT '04*, Springer-Verlag, 2004. Full paper available from the I CR eprint archive at `http://eprint.iacr.org/2004/193/`.

ppendices

GCM for 64-bit block ciphers

While GCM is clearly designed for use with ES, it may be desirable to use it in applications that require block ciphers with 64-bit blocks. Unlike other dual-use modes that are based on counter mode, the way that GCM processes its nonce lends itself to practical implementations in such an environment: the same IV format can be used with both the 128-bit and 64-bit versions. In this section, we define GCM for 64-bit block ciphers, which has some slight but important differences. The notations used in this section follow that of Section 2, with the exception that the blocks P_i, C_i, and A_i are 64 bits long. The length u of the final blocks P_n^* and C_n^* is no greater than 64 bits, as is the length v of the final block A_n^*.

GH SH has several minor differences. First, it uses a field polynomial appropriate to the block size: $1 + \quad + \quad ^3 + \quad ^4 + \quad ^{64}$. Second, the way that the lengths are handled is slightly different. GH SH64 $H, A, C)$ is defined by the following equations:

$$X_i = \begin{cases} 0 & \text{for } i = 0 \\ X_{i\quad 1} \oplus A_i) \cdot H & \text{for } i = 1, \ldots, m \quad 1 \\ X_{m\quad 1} \oplus \quad A_m^* \| 0^{64\quad v})) \cdot H & \text{for } i = m \\ X_{i\quad 1} \oplus C_i) \cdot H & \text{for } i = m+1, \ldots, m+n \quad 1 \\ X_{m+n\quad 1} \oplus \quad C_m^* \| 0^{64\quad u})) \cdot H & \text{for } i = m+n \\ \quad X_{m+n} \oplus \text{len} \quad A)) \cdot H \oplus \text{len} \quad C)) \cdot H & \text{for } i = m+n+1. \end{cases}$$

The authenticated encryption operation for 64-bit ciphers is defined by the following equations:

$$\begin{aligned} H &= E \quad K, 0^{64}) \\ Y_0 &= \begin{matrix} 0^{31} 1 \| IV & \text{if len } \quad IV) = 32 \\ \text{GH SH64} \quad H, \{\}, IV) & \text{otherwise.} \end{matrix} \\ Y_i &= \text{incr} \quad Y_{i\quad 1}) \text{ for } i = 1, \ldots, n \\ C_i &= P_i \oplus E \quad K, Y_i) \text{ for } i = 1, \ldots, n \quad 1 \\ C_n &= P_n \oplus \text{MSB}_u \quad E \quad K, Y_n)) \\ T &= \text{MSB}_t \text{ GH SH64} \quad H, A, C) \oplus E \quad K, Y_0)) \end{aligned}$$

The equations for the authenticated decryption operation for 64-bit ciphers are:

$$H = E\ \ K, 0^{64})$$

$$Y_0 = \begin{array}{ll} 0^{31}1\|IV & \text{if len } \ IV) = 32 \\ \text{GH}\quad \text{SH64}\ \ H, \{\}, IV) & \text{otherwise.} \end{array}$$

$$T' = \text{MSB}_t\ \text{GH}\quad \text{SH64}\ \ H, A, C) \oplus E\ \ K, Y_0))$$

$$Y_i = \text{incr}\ \ Y_{i\quad 1}) \text{ for } i = 1, \ldots, n$$

$$P_i = C_i \oplus E\ \ K, Y_i) \text{ for } i = 1, \ldots, n$$

$$P_n = C_n \oplus \text{MSB}_u\ \ E\ \ K, Y_n))$$

B ES Test Vectors

This appendix contains test cases for ES GCM, with ES key sizes of 128, 192, and 256 bits. These cases use the same notation as in Equations 1 and 2, with the exception that N_i is used in place of X_i when GH SH is used to compute Y_0, in order to distinguish that case from the later invocation of GH SH. ll values are in hexadecimal, and a zero-length variable is indicated by the absence of any hex digits. Each line consists of 128 bits of data, and variables whose lengths exceed that value are continued on successive lines. The leftmost hex digit corresponds to the leftmost four bits of the variable. For example, the lowest 128 bits of the field polynomial are represented as `e1000000000000000000000000000000`.

	Variable	Value
	K	`00000000000000000000000000000000`
	P	
	IV	`000000000000000000000000`
	H	`66e94bd4ef8a2c3b884cfa59ca342b2e`
Test Case 1	Y_0	`00000000000000000000000000000001`
	E $K, Y_0)$	`58e2fccefa7e3061367f1d57a4e7455a`
	len $A)\|\|$len $C)$	`00000000000000000000000000000000`
	GH SH $H, A, C)$	`00000000000000000000000000000000`
	C	
	T	`58e2fccefa7e3061367f1d57a4e7455a`

	Variable	Value
	K	`00000000000000000000000000000000`
	P	`00000000000000000000000000000000`
	IV	`000000000000000000000000`
	H	`66e94bd4ef8a2c3b884cfa59ca342b2e`
	Y_0	`00000000000000000000000000000001`
Test Case 2	E $K, Y_0)$	`58e2fccefa7e3061367f1d57a4e7455a`
	Y_1	`00000000000000000000000000000002`
	E $K, Y_1)$	`0388dace60b6a392f328c2b971b2fe78`
	X_1	`5e2ec746917062882c85b0685353deb7`
	len $A)\|\|$len $C)$	`00000000000000000000000000000080`
	GH SH $H, A, C)$	`f38cbb1ad69223dcc3457ae5b6b0f885`
	C	`0388dace60b6a392f328c2b971b2fe78`
	T	`ab6e47d42cec13bdf53a67b21257bddf`

Test Case 3

Variable	Value
K	feffe9928665731c6d6a8f9467308308
P	d9313225f88406e5a55909c5aff5269a
	86a7a9531534f7da2e4c303d8a318a72
	1c3c0c95956809532fcf0e2449a6b525
	b16aedf5aa0de657ba637b391aafd255
IV	cafebabefacedbaddecaf888
H	b83b533708bf535d0aa6e52980d53b78
Y_0	cafebabefacedbaddecaf88800000001
$E\ K, Y_0)$	3247184b3c4f69a44dbcd22887bbb418
Y_1	cafebabefacedbaddecaf88800000002
$E\ K, Y_1)$	9bb22ce7d9f372c1ee2b28722b25f206
Y_2	cafebabefacedbaddecaf88800000003
$E\ K, Y_2)$	650d887c3936533a1b8d4e1ea39d2b5c
Y_3	cafebabefacedbaddecaf88800000004
$E\ K, Y_3)$	3de91827c10e9a4f5240647ee5221f20
Y_4	cafebabefacedbaddecaf88800000005
$E\ K, Y_4)$	aac9e6ccc0074ac0873b9ba85d908bd0
X_1	59ed3f2bb1a0aaa07c9f56c6a504647b
X_2	b714c9048389afd9f9bc5c1d4378e052
X_3	47400c6577b1ee8d8f40b2721e86ff10
X_4	4796cf49464704b5dd91f159bb1b7f95
len $A)\|\|$len $C)$	00000000000000000000000000000200
GH SH $H, A, C)$	7f1b32b81b820d02614f8895ac1d4eac
C	42831ec2217774244b7221b784d0d49c
	e3aa212f2c02a4e035c17e2329aca12e
	21d514b25466931c7d8f6a5aac84aa05
	1ba30b396a0aac973d58e091473f5985
T	4d5c2af327cd64a62cf35abd2ba6fab4

Test Case 4

Variable	Value
K	`feffe9928665731c6d6a8f9467308308`
P	`d9313225f88406e5a55909c5aff5269a`
	`86a7a9531534f7da2e4c303d8a318a72`
	`1c3c0c95956809532fcf0e2449a6b525`
	`b16aedf5aa0de657ba637b39`
A	`feedfacedeadbeeffeedfacedeadbeef`
	`abaddad2`
IV	`cafebabefacedbaddecaf888`
H	`b83b533708bf535d0aa6e52980d53b78`
Y_0	`cafebabefacedbaddecaf88800000001`
$E\ K, Y_0)$	`3247184b3c4f69a44dbcd22887bbb418`
X_1	`ed56aaf8a72d67049fdb9228edba1322`
X_2	`cd47221ccef0554ee4bb044c88150352`
Y_1	`cafebabefacedbaddecaf88800000002`
$E\ K, Y_1)$	`9bb22ce7d9f372c1ee2b28722b25f206`
Y_2	`cafebabefacedbaddecaf88800000003`
$E\ K, Y_2)$	`650d887c3936533a1b8d4e1ea39d2b5c`
Y_3	`cafebabefacedbaddecaf88800000004`
$E\ K, Y_3)$	`3de91827c10e9a4f5240647ee5221f20`
Y_4	`cafebabefacedbaddecaf88800000005`
$E\ K, Y_4)$	`aac9e6ccc0074ac0873b9ba85d908bd0`
X_3	`54f5e1b2b5a8f9525c23924751a3ca51`
X_4	`324f585c6ffc1359ab371565d6c45f93`
X_5	`ca7dd446af4aa70cc3c0cd5abba6aa1c`
X_6	`1590df9b2eb6768289e57d56274c8570`
len $A)\|\|$len $C)$	`00000000000000a000000000000001e0`
GH SH $H, A, C)$	`698e57f70e6ecc7fd9463b7260a9ae5f`
C	`42831ec2217774244b7221b784d0d49c`
	`e3aa212f2c02a4e035c17e2329aca12e`
	`21d514b25466931c7d8f6a5aac84aa05`
	`1ba30b396a0aac973d58e091`
T	`5bc94fbc3221a5db94fae95ae7121a47`

Test Case 5

Variable	Value
K	feffe9928665731c6d6a8f9467308308
P	d9313225f88406e5a55909c5aff5269a
	86a7a9531534f7da2e4c303d8a318a72
	1c3c0c95956809532fcf0e2449a6b525
	b16aedf5aa0de657ba637b39
A	feedfacedeadbeeffeedfacedeadbeef
	abaddad2
IV	cafebabefacedbad
H	b83b533708bf535d0aa6e52980d53b78
N_1	6f288b846e5fed9a18376829c86a6a16
len $\{\})\|\|$len $IV)$	00000000000000000000000000000040
Y_0	c43a83c4c4badec4354ca984db252f7d
$E\ \ K, Y_0)$	e94ab9535c72bea9e089c93d48e62fb0
X_1	ed56aaf8a72d67049fdb9228edba1322
X_2	cd47221ccef0554ee4bb044c88150352
Y_1	c43a83c4c4badec4354ca984db252f7e
$E\ \ K, Y_1)$	b8040969d08295afd226fcda0ddf61cf
Y_2	c43a83c4c4badec4354ca984db252f7f
$E\ \ K, Y_2)$	ef3c83225af93122192ad5c4f15dfe51
Y_3	c43a83c4c4badec4354ca984db252f80
$E\ \ K, Y_3)$	6fbc659571f72de104c67b609d2fde67
Y_4	c43a83c4c4badec4354ca984db252f81
$E\ \ K, Y_4)$	f8e3581441a1e950785c3ea1430c6fa6
X_3	9379e2feae14649c86cf2250e3a81916
X_4	65dde904c92a6b3db877c4817b50a5f4
X_5	48c53cf863b49a1b0bbfc48c3baaa89d
X_6	08c873f1c8cec3effc209a07468caab1
len $A)\|\|$len $C)$	00000000000000a000000000000001e0
GH SH $H, A, C)$	df586bb4c249b92cb6922877e444d37b
C	61353b4c2806934a777ff51fa22a4755
	699b2a714fcdc6f83766e5f97b6c7423
	73806900e49f24b22b097544d4896b42
	4989b5e1ebac0f07c23f4598
T	3612d2e79e3b0785561be14aaca2fccb

Test Case 6

Variable	Value
K	`feffe9928665731c6d6a8f9467308308`
P	`d9313225f88406e5a55909c5aff5269a`
	`86a7a9531534f7da2e4c303d8a318a72`
	`1c3c0c95956809532fcf0e2449a6b525`
	`b16aedf5aa0de657ba637b39`
A	`feedfacedeadbeeffeedfacedeadbeef`
	`abaddad2`
IV	`9313225df88406e555909c5aff5269aa`
	`6a7a9538534f7da1e4c303d2a318a728`
	`c3c0c95156809539fcf0e2429a6b5254`
	`16aedbf5a0de6a57a637b39b`
H	`b83b533708bf535d0aa6e52980d53b78`
N_1	`004d6599d7fb1634756e1e299d81630f`
N_2	`88ffe8a3c8033df4b54d732f7f88408e`
N_3	`24e694cfab657beabba8055aad495e23`
N_4	`d8349a5eda24943c8fbb2ef5168b20cb`
len $\{\})\|\|$len $IV)$	`000000000000000000000000000001e0`
Y_0	`3bab75780a31c059f83d2a44752f9864`
$E\ K, Y_0)$	`7dc63b399f2d98d57ab073b6baa4138e`
X_1	`ed56aaf8a72d67049fdb9228edba1322`
X_2	`cd47221ccef0554ee4bb044c88150352`
Y_1	`3bab75780a31c059f83d2a44752f9865`
$E\ K, Y_1)$	`55d37bbd9ad21353a6f93a690eca9e0e`
Y_2	`3bab75780a31c059f83d2a44752f9866`
$E\ K, Y_2)$	`3836bbf6d696e672946a1a01404fa6d5`
Y_3	`3bab75780a31c059f83d2a44752f9867`
$E\ K, Y_3)$	`1dd8a5316ecc35c3e313bca59d2ac94a`
Y_4	`3bab75780a31c059f83d2a44752f9868`
$E\ K, Y_4)$	`6742982706a9f154f657d5dc94b746db`
X_3	`31727669c63c6f078b5d22adbbbca384`
X_4	`480c00db2679065a7ed2f771a53acacd`
X_5	`1c1ae3c355e2214466a9923d2ba6ab35`
X_6	`0694c6f16bb0275a48891d06590344b0`
len $A)\|\|$len $C)$	`00000000000000a000000000000001e0`
GH SH $H, A, C)$	`1c5afe9760d3932f3c9a878aac3dc3de`
C	`8ce24998625615b603a033aca13fb894`
	`be9112a5c3a211a8ba262a3cca7e2ca7`
	`01e4a9a4fba43c90ccdcb281d48c7c6f`
	`d62875d2aca417034c34aee5`
T	`619cc5aefffe0bfa462af43c1699d050`

Test Case 7

Variable	Value
K	00000000000000000000000000000000 0000000000000000
P	
IV	000000000000000000000000
H	aae06992acbf52a3e8f4a96ec9300bd7
Y_0	00000000000000000000000000000001
$E\ K, Y_0)$	cd33b28ac773f74ba00ed1f312572435
len $A)\|\|$len $C)$	00000000000000000000000000000000
GH SH $H, A, C)$	00000000000000000000000000000000
C	
T	cd33b28ac773f74ba00ed1f312572435

Test Case 8

Variable	Value
K	00000000000000000000000000000000 0000000000000000
P	00000000000000000000000000000000
IV	000000000000000000000000
H	aae06992acbf52a3e8f4a96ec9300bd7
Y_0	00000000000000000000000000000001
$E\ K, Y_0)$	cd33b28ac773f74ba00ed1f312572435
Y_1	00000000000000000000000000000002
$E\ K, Y_1)$	98e7247c07f0fe411c267e4384b0f600
X_1	90e87315fb7d4e1b4092ec0cbfda5d7d
len $A)\|\|$len $C)$	00000000000000000000000000000080
GH SH $H, A, C)$	e2c63f0ac44ad0e02efa05ab6743d4ce
C	98e7247c07f0fe411c267e4384b0f600
T	2ff58d80033927ab8ef4d4587514f0fb

Test Case 9

Variable	Value
K	`feffe9928665731c6d6a8f9467308308`
	`feffe9928665731c`
P	`d9313225f88406e5a55909c5aff5269a`
	`86a7a9531534f7da2e4c303d8a318a72`
	`1c3c0c95956809532fcf0e2449a6b525`
	`b16aedf5aa0de657ba637b391aafd255`
IV	`cafebabefacedbaddecaf888`
H	`466923ec9ae682214f2c082badb39249`
Y_0	`cafebabefacedbaddecaf88800000001`
$E\ K, Y_0)$	`c835aa88aebbc94f5a02e179fdcfc3e4`
Y_1	`cafebabefacedbaddecaf88800000002`
$E\ K, Y_1)$	`e0b1f82ec484eea44e5ff30128df01cd`
Y_2	`cafebabefacedbaddecaf88800000003`
$E\ K, Y_2)$	`0339b5b9b3db2e5e4cc9a38986906bee`
Y_3	`cafebabefacedbaddecaf88800000004`
$E\ K, Y_3)$	`614b3195542ccc7683ae933c81ec8a62`
Y_4	`cafebabefacedbaddecaf88800000005`
$E\ K, Y_4)$	`a988a97e85eec28e76b95c29b6023003`
X_1	`dddca3f91c17821ffac4a6d0fed176f7`
X_2	`a4e84ac60e2730f4a7e0e1eef708b198`
X_3	`e67592048dd7153973a0dbbb8804bee2`
X_4	`503e86628536625fb746ce3cecea433f`
len $A)\|\|$len $C)$	`00000000000000000000000000000200`
GH SH $H, A, C)$	`51110d40f6c8fff0eb1ae33445a889f0`
C	`3980ca0b3c00e841eb06fac4872a2757`
	`859e1ceaa6efd984628593b40ca1e19c`
	`7d773d00c144c525ac619d18c84a3f47`
	`18e2448b2fe324d9ccda2710acade256`
T	`9924a7c8587336bfb118024db8674a14`

Test Case 10

Variable	Value
K	`feffe9928665731c6d6a8f9467308308`
	`feffe9928665731c`
P	`d9313225f88406e5a55909c5aff5269a`
	`86a7a9531534f7da2e4c303d8a318a72`
	`1c3c0c95956809532fcf0e2449a6b525`
	`b16aedf5aa0de657ba637b39`
A	`feedfacedeadbeeffeedfacedeadbeef`
	`abaddad2`
IV	`cafebabefacedbaddecaf888`
H	`466923ec9ae682214f2c082badb39249`
Y_0	`cafebabefacedbaddecaf88800000001`
$E \; K,Y_0)$	`c835aa88aebbc94f5a02e179fdcfc3e4`
X_1	`f3bf7ba3e305aeb05ed0d2e4fe076666`
X_2	`20a51fa2302e9c01b87c48f2c3d91a56`
Y_1	`cafebabefacedbaddecaf88800000002`
$E \; K,Y_1)$	`e0b1f82ec484eea44e5ff30128df01cd`
Y_2	`cafebabefacedbaddecaf88800000003`
$E \; K,Y_2)$	`0339b5b9b3db2e5e4cc9a38986906bee`
Y_3	`cafebabefacedbaddecaf88800000004`
$E \; K,Y_3)$	`614b3195542ccc7683ae933c81ec8a62`
Y_4	`cafebabefacedbaddecaf88800000005`
$E \; K,Y_4)$	`a988a97e85eec28e76b95c29b6023003`
X_3	`714f9700ddf520f20695f6180c6e669d`
X_4	`e858680b7b240d2ecf7e06bbad4524e2`
X_5	`3f4865abd6bb3fb9f5c4a816f0a9b778`
X_6	`4256f67fe87b4f49422ba11af857c973`
len $A)$||len $C)$	`0000000000000a0000000000000001e0`
GH SH $H,A,C)$	`ed2ce3062e4a8ec06db8b4c490e8a268`
C	`3980ca0b3c00e841eb06fac4872a2757`
	`859e1ceaa6efd984628593b40ca1e19c`
	`7d773d00c144c525ac619d18c84a3f47`
	`18e2448b2fe324d9ccda2710`
T	`2519498e80f1478f37ba55bd6d27618c`

Test Case 11

Variable	Value
K	feffe9928665731c6d6a8f9467308308
	feffe9928665731c
P	d9313225f88406e5a55909c5aff5269a
	86a7a9531534f7da2e4c303d8a318a72
	1c3c0c95956809532fcf0e2449a6b525
	b16aedf5aa0de657ba637b39
A	feedfacedeadbeeffeedfacedeadbeef
	abaddad2
IV	cafebabefacedbad
H	466923ec9ae682214f2c082badb39249
N_1	9473c07b02544299cf007c42c5778218
len $\{\})$||len $IV)$	00000000000000000000000000000040
Y_0	a14378078d27258a6292737e1802ada5
$E\ \ K, Y_0)$	7bb6d647c902427ce7cf26563a337371
X_1	f3bf7ba3e305aeb05ed0d2e4fe076666
X_2	20a51fa2302e9c01b87c48f2c3d91a56
Y_1	a14378078d27258a6292737e1802ada6
$E\ \ K, Y_1)$	d621c7bc5690a7b1487dbaab8ac76b22
Y_2	a14378078d27258a6292737e1802ada7
$E\ \ K, Y_2)$	43c1ca7de78f4495ad0b18324e61fa25
Y_3	a14378078d27258a6292737e1802ada8
$E\ \ K, Y_3)$	e1e0254a0f2f1626e9aa4ff09d7c64ec
Y_4	a14378078d27258a6292737e1802ada9
$E\ \ K, Y_4)$	5850f4502486a1681a9319ce7d0afa59
X_3	8bdedafd6ee8e529689de3a269b8240d
X_4	6607feb377b49c9ecdbc696344fe22d8
X_5	8a19570a06500ba9405fcece4a73fb48
X_6	8532826e63ce4a5b89b70fa28f8070fe
len $A)$||len $C)$	00000000000000a000000000000001e0
GH SH $H, A, C)$	1e6a133806607858ee80eaf237064089
C	0f10f599ae14a154ed24b36e25324db8
	c566632ef2bbb34f8347280fc4507057
	fddc29df9a471f75c66541d4d4dad1c9
	e93a19a58e8b473fa0f062f7
T	65dcc57fcf623a24094fcca40d3533f8

Test Case 12

Variable	Value
K	feffe9928665731c6d6a8f9467308308
	feffe9928665731c
P	d9313225f88406e5a55909c5aff5269a
	86a7a9531534f7da2e4c303d8a318a72
	1c3c0c95956809532fcf0e2449a6b525
	b16aedf5aa0de657ba637b39
A	feedfacedeadbeeffeedfacedeadbeef
	abaddad2
IV	9313225df88406e555909c5aff5269aa
	6a7a9538534f7da1e4c303d2a318a728
	c3c0c95156809539fcf0e2429a6b5254
	16aedbf5a0de6a57a637b39b
H	466923ec9ae682214f2c082badb39249
N_1	19aef0f04763b0c87903c5a217d5314f
N_2	62120253f79efc978625d1feb03b5b5b
N_3	b6ce2a84e366de900fa78a1653df77fb
N_4	374ecad90487f0bb261ba817447e022c
len $\{\})\|\|$len $IV)$	000000000000000000000000000001e0
Y_0	4505cdc367a054c5002820e96aebef27
$E\ K, Y_0)$	5ea3194f9dd012a3b9bc5103d6e0284d
X_1	f3bf7ba3e305aeb05ed0d2e4fe076666
X_2	20a51fa2302e9c01b87c48f2c3d91a56
Y_1	4505cdc367a054c5002820e96aebef28
$E\ K, Y_1)$	0b4fba4de46722d9ed691f9f2029df65
Y_2	4505cdc367a054c5002820e96aebef29
$E\ K, Y_2)$	9b4e088bf380b03540bb87a5a257e437
Y_3	4505cdc367a054c5002820e96aebef2a
$E\ K, Y_3)$	9ddb9c873a5cd48acd3f397cd28f9896
Y_4	4505cdc367a054c5002820e96aebef2b
$E\ K, Y_4)$	5716ee92eff7c4b053d44c0294ea88cd
X_3	f70d61693ea7f53f08c866d6eedb1e4b
X_4	dc40bc9a181b35aed66488071ef282ae
X_5	85ffa424b87b35cac7be9c450f0d7aee
X_6	65233cbe5251f7d246bfc967a8678647
len $A)\|\|$len $C)$	00000000000000a000000000000001e0
GH SH $H, A, C)$	82567fb0b4cc371801eadec005968e94
C	d27e88681ce3243c4830165a8fdcf9ff
	1de9a1d8e6b447ef6ef7b79828666e45
	81e79012af34ddd9e2f037589b292db3
	e67c036745fa22e7e9b7373b
T	dcf566ff291c25bbb8568fc3d376a6d9

	Variable	Value
	K	00000000000000000000000000000000
		00000000000000000000000000000000
	P	
	IV	000000000000000000000000
	H	dc95c078a2408989ad48a21492842087
Test Case 13	Y_0	00000000000000000000000000000001
	$E\ K, Y_0)$	530f8afbc74536b9a963b4f1c4cb738b
	len $A)$||len $C)$	00000000000000000000000000000000
	GH SH $H, A, C)$	00000000000000000000000000000000
	C	
	T	530f8afbc74536b9a963b4f1c4cb738b

	Variable	Value
	K	00000000000000000000000000000000
		00000000000000000000000000000000
	P	00000000000000000000000000000000
	IV	000000000000000000000000
	H	dc95c078a2408989ad48a21492842087
	Y_0	00000000000000000000000000000001
Test Case 14	$E\ K, Y_0)$	530f8afbc74536b9a963b4f1c4cb738b
	Y_1	00000000000000000000000000000002
	$E\ K, Y_1)$	cea7403d4d606b6e074ec5d3baf39d18
	X_1	fd6ab7586e556dba06d69cfe6223b262
	len $A)$||len $C)$	00000000000000000000000000000080
	GH SH $H, A, C)$	83de425c5edc5d498f382c441041ca92
	C	cea7403d4d606b6e074ec5d3baf39d18
	T	d0d1c8a799996bf0265b98b5d48ab919

Test Case 15

Variable	Value
K	feffe9928665731c6d6a8f9467308308
	feffe9928665731c6d6a8f9467308308
P	d9313225f88406e5a55909c5aff5269a
	86a7a9531534f7da2e4c303d8a318a72
	1c3c0c95956809532fcf0e2449a6b525
	b16aedf5aa0de657ba637b391aafd255
IV	cafebabefacedbaddecaf888
H	acbef20579b4b8ebce889bac8732dad7
Y_0	cafebabefacedbaddecaf88800000001
$E\ \ K, Y_0)$	fd2caa16a5832e76aa132c1453eeda7e
Y_1	cafebabefacedbaddecaf88800000002
$E\ \ K, Y_1)$	8b1cf3d561d27be251263e66857164e7
Y_2	cafebabefacedbaddecaf88800000003
$E\ \ K, Y_2)$	e29d258faad137135bd49280af645bd8
Y_3	cafebabefacedbaddecaf88800000004
$E\ \ K, Y_3)$	908c82ddcc65b26e887f85341f243d1d
Y_4	cafebabefacedbaddecaf88800000005
$E\ \ K, Y_4)$	749cf39639b79c5d06aa8d5b932fc7f8
X_1	fcbefb78635d598eddaf982310670f35
X_2	29de812309d3116a6eff7ec844484f3e
X_3	45fad9deeda9ea561b8f199c3613845b
X_4	ed95f8e164bf3213febc740f0bd9c6af
len $A)$||len $C)$	00000000000000000000000000000200
GH SH $H, A, C)$	4db870d37cb75fcb46097c36230d1612
C	522dc1f099567d07f47f37a32a84427d
	643a8cdcbfe5c0c97598a2bd2555d1aa
	8cb08e48590dbb3da7b08b1056828838
	c5f61e6393ba7a0abcc9f662898015ad
T	b094dac5d93471bdec1a502270e3cc6c

Test Case 16

Variable	Value
K	feffe9928665731c6d6a8f9467308308
	feffe9928665731c6d6a8f9467308308
P	d9313225f88406e5a55909c5aff5269a
	86a7a9531534f7da2e4c303d8a318a72
	1c3c0c95956809532fcf0e2449a6b525
	b16aedf5aa0de657ba637b39
A	feedfacedeadbeeffeedfacedeadbeef
	abaddad2
IV	cafebabefacedbaddecaf888
H	acbef20579b4b8ebce889bac8732dad7
Y_0	cafebabefacedbaddecaf88800000001
$E\ \ K, Y_0)$	fd2caa16a5832e76aa132c1453eeda7e
X_1	5165d242c2592c0a6375e2622cf925d2
X_2	8efa30ce83298b85fe71abefc0cdd01d
Y_1	cafebabefacedbaddecaf88800000002
$E\ \ K, Y_1)$	8b1cf3d561d27be251263e66857164e7
Y_2	cafebabefacedbaddecaf88800000003
$E\ \ K, Y_2)$	e29d258faad137135bd49280af645bd8
Y_3	cafebabefacedbaddecaf88800000004
$E\ \ K, Y_3)$	908c82ddcc65b26e887f85341f243d1d
Y_4	cafebabefacedbaddecaf88800000005
$E\ \ K, Y_4)$	749cf39639b79c5d06aa8d5b932fc7f8
X_3	abe07e0bb62354177480b550f9f6cdcc
X_4	3978e4f141b95f3b4699756b1c3c2082
X_5	8abf3c48901debe76837d8a05c7d6e87
X_6	9249beaf520c48b912fa120bbf391dc8
len $A)$||len $C)$	0000000000000a0000000000000001e0
GH SH $H, A, C)$	8bd0c4d8aacd391e67cca447e8c38f65
C	522dc1f099567d07f47f37a32a84427d
	643a8cdcbfe5c0c97598a2bd2555d1aa
	8cb08e48590dbb3da7b08b1056828838
	c5f61e6393ba7a0abcc9f662
T	76fc6ece0f4e1768cddf8853bb2d551b

Test Case 17

Variable	Value
K	feffe9928665731c6d6a8f9467308308
	feffe9928665731c6d6a8f9467308308
P	d9313225f88406e5a55909c5aff5269a
	86a7a9531534f7da2e4c303d8a318a72
	1c3c0c95956809532fcf0e2449a6b525
	b16aedf5aa0de657ba637b39
A	feedfacedeadbeeffeedfacedeadbeef
	abaddad2
IV	cafebabefacedbad
H	acbef20579b4b8ebce889bac8732dad7
N_1	90c22e3d2aca34b971e8bd09708fae5c
$\text{len}\ \{\})\|\|\text{len}\ IV)$	00000000000000000000000000000040
Y_0	0095df49dd90abe3e4d252475748f5d4
$E\ \ K, Y_0)$	4f903f37fe611d454217fbfa5cd7d791
X_1	5165d242c2592c0a6375e2622cf925d2
X_2	8efa30ce83298b85fe71abefc0cdd01d
Y_1	0095df49dd90abe3e4d252475748f5d5
$E\ \ K, Y_1)$	1a471fd432fc7bd70b1ec8fe5e6d6251
Y_2	0095df49dd90abe3e4d252475748f5d6
$E\ \ K, Y_2)$	29bd481e1ea39d20eb63c7ea118b1792
Y_3	0095df49dd90abe3e4d252475748f5d7
$E\ \ K, Y_3)$	e2898e46ac5cada3ba83cc1272618a5d
Y_4	0095df49dd90abe3e4d252475748f5d8
$E\ \ K, Y_4)$	d3c6aefbcea602ce4e1fe026065447bf
X_3	55e1ff68f9249e64b95223858e5cb936
X_4	cef1c034383dc96f733aaa4c99bd3e61
X_5	68588d004fd468f5854515039b08165d
X_6	2378943c034697f72a80fce5059bf3f3
$\text{len}\ A)\|\|\text{len}\ C)$	00000000000000a000000000000001e0
$\text{GH}\ \ \ \text{SH}\ H, A, C)$	75a34288b8c68f811c52b2e9a2f97f63
C	c3762df1ca787d32ae47c13bf19844cb
	af1ae14d0b976afac52ff7d79bba9de0
	feb582d33934a4f0954cc2363bc73f78
	62ac430e64abe499f47c9b1f
T	3a337dbf46a792c45e454913fe2ea8f2

Test Case 18

Variable	Value
K	feffe9928665731c6d6a8f9467308308
	feffe9928665731c6d6a8f9467308308
P	d9313225f88406e5a55909c5aff5269a
	86a7a9531534f7da2e4c303d8a318a72
	1c3c0c95956809532fcf0e2449a6b525
	b16aedf5aa0de657ba637b39
A	feedfacedeadbeeffeedfacedeadbeef
	abaddad2
IV	9313225df88406e555909c5aff5269aa
	6a7a9538534f7da1e4c303d2a318a728
	c3c0c95156809539fcf0e2429a6b5254
	16aedbf5a0de6a57a637b39b
H	acbef20579b4b8ebce889bac8732dad7
N_1	0bfe66e2032f195516379f5fb710f987
N_2	f0631554d11409915feec8f9f5102aba
N_3	749b90dda19a1557fd9e9fd31fed1d14
N_4	7a6a833f260d848793b327cb07d1b190
len $\{\})\|\|$len $IV)$	000000000000000000000000000001e0
Y_0	0cd953e2140a5976079f8e2406bc8eb4
$E\ K, Y_0)$	71b54d092bb0c3d9ba94538d4096e691
X_1	5165d242c2592c0a6375e2622cf925d2
X_2	8efa30ce83298b85fe71abefc0cdd01d
Y_1	0cd953e2140a5976079f8e2406bc8eb5
$E\ K, Y_1)$	83bcdd0af41a551452047196ca6b0cba
Y_2	0cd953e2140a5976079f8e2406bc8eb6
$E\ K, Y_2)$	68151b79baea93c38e149b72e545e186
Y_3	0cd953e2140a5976079f8e2406bc8eb7
$E\ K, Y_3)$	13fccf22159a4d16026ce5d58c7e99fb
Y_4	0cd953e2140a5976079f8e2406bc8eb8
$E\ K, Y_4)$	132b64628a031e79fecd050675a64f07
X_3	e963941cfa8c417bdaa3b3d94ab4e905
X_4	2178d7f836e5fa105ce0fdf0fc8f0654
X_5	bac14eeba3216f966b3e7e011475b832
X_6	cc9ae9175729a649936e890bd971a8bf
len $A)\|\|$len $C)$	00000000000000a000000000000001e0
GH SH $H, A, C)$	d5ffcf6fc5ac4d69722187421a7f170b
C	5a8def2f0c9e53f1f75d7853659e2a20
	eeb2b22aafde6419a058ab4f6f746bf4
	0fc0c3b780f244452da3ebf1c5d82cde
	a2418997200ef82e44ae7e3f
T	a44a8266ee1c8eb0c8b5d4cf5ae9f19a