Comments to NIST concerning AES Modes of Operation:

PMAC: A Parallelizable Message Authentication Code

Phillip Rogaway

University of California at Davis (USA) and

Chiang Mai University (Thailand)

rogaway@cs.ucdavis.edu

www.cs.ucdavis.edu/∼rogaway

Preliminary Draft: October 16, 2000

Abstract

We describe a MAC (message authentication code) which is deterministic, parallelizable, and uses only $\lceil |M|/n \rceil$ block-cipher invocations to MAC a non-empty string M (where n is the blocksize of the underlying block cipher). The MAC can be proven secure (work to appear) in the reduction-based approach of modern cryptography. The MAC is similar to one recently suggested by Gligor and Donescu [5].

1 Introduction

PMAC and its characteristics This note describes a new message authentication code, PMAC. Unlike customary modes for message authentication, the construction here is fully parallelizable. This will result in faster authentication in a variety of settings.

The PMAC construction is stingy in its use of block-cipher calls, employing just $\lceil |M|/n \rceil$ block-cipher invocations to MAC a nonempty string M using an n-bit block cipher.

A MAC computed by PMAC can have any length from up to n bits.

Unlike the CBC MAC (in its basic form), PMAC can be applied to any message M; in particular, $|M|$ need not be a positive multiple of n. Likewise, messages being MACed do not need to be of one fixed length; messages of varying lengths can be safely MACed.

When using PMAC with AES, the key for PMAC is a single AES key, and all AES invocations are under that key.

The name PMAC is intended to suggest Parallelizable MAC.

Security Assuming that the underlying block cipher behaves as a pseudorandom permutation, PMAC achieves existential unforgeability under an adaptive chosen-message attack, the now standard notion of security for MACs [6, 3]. In joint work with Mihir Bellare and John Black, a proof to this effect is currently being prepared. Since this proof and its writeup is not complete, the current algorithm should be considered provisional.

Prior work The PMAC construction is similar to the XOR MAC of Bellare, Guérin, and Rogaway [2] and the variant of this defined by Bernstein [4]. It is even more similar to the XECB MAC of Gligor and Donescu [5]. The latter work inspired our own. But PMAC is more efficient than either alternative. In particular, The XOR MAC is stateful or randomized, and it uses a constant fraction more block-cipher invocations than PMAC. The XECB MAC is stateful or randomized, and it uses one more block-cipher invocation. In either case, the consequence of being stateful or randomized is that the MAC itself is longer, since it must include a counter or random number. Being stateful or randomized also presents added operational difficulties—the sender needs a source of random bits, or needs to maintain a counter across MAC invocations.

It is possible to view PMAC as (an optimized version of) yet another MAC construction falling under the Carter-Wegman paradigm [8]. Other MACs which fall under that paradigm are parallelizable if they employ a parallelizable universal hash-function family. But specifying and implementing a (non-cryptographic) universal hash function (particularly a fast one) is more involved than the simple mechanism we give here.

2 Notation

Fix a block cipher E which enciphers an n-bit string X using a k-bit key K, obtaining ciphertext block $Y = E_K(X)$. For $E = \text{AES}$ we have $n = 128$ and $k \in \{128, 192, 256\}$.

The authentication tags we specify can have any number of bits, $tagLen$, from 1 to n. A standard should allow such tag-truncation since tags in excess of 80 bits, say, utilize extra bits but provide no meaningful increment to security. A default value of $tagLen = 64$ is probably good.

By 0^i and 1^i we mean strings of i 0's and 1's, respectively. For A a string of length less than n, by $pad_n(A)$ we mean the string $0^{n-|A|-1}1\,A$: that is, prepend 0-bits and then a 1-bit so as to get to length n. (Appending a 1-bit and then 0-bits would also be fine.)

If A is a binary string then $|A|$ denotes its length, in bits. If A and B are strings then AB denotes their concatenation. If A and B are strings of equal length then $A \oplus B$ is their bitwise XOR and $A \vee B$ is their bitwise OR and $A \wedge B$ is their bitwise AND. By $A[\text{bit } i]$ we mean the i-th bit of A (regarded, where necessary, as the number 0 or the number 1), where characters are numbered left-to-right, starting at 1. By $A[\text{bits } \ell \text{ to } r]$ we mean $A[\text{bit } \ell]A[\text{bit } \ell+1] \cdots A[\text{bit } r]$.

3 The PMAC Algorithm (in general, and PMAC/add)

Addition, multiplication, and $\text{Final}(\cdot)$ We assume an addition operator $+$ from $\{0,1\}^n \times \{0,1\}^n$ to $\{0,1\}^n$ and a multiplication operator (with no explicitly written symbol) from $\{1, 2, 3, \ldots, \} \times \{0,1\}^n$ to $\{0,1\}^n$. We also assume a map $\text{Final} : \{0,1\}^n \to \{0,1\}^n$. For concreteness, we now give these functions a particular instantiation. Later we will revise this instantiation to demonstrate a couple of further possibilities.

PMAC/add For the addition modulo 2^n version of PMAC, PMAC/add, instantiate $+$ by computer addition of n-bit words (ignoring any carry) and instantiate iL, for $i \geq 1$, by repeated addition. Let $\text{Final}(L)$ be $\overline{L}$, the bitwise complement of L.

(A more formal definition follows. Let $A, B \in \{0,1\}^n$. By $\mathsf{str2num}(A)$ we mean the nonnegative integer that is represented by A, that is, $\sum_{i=1}^{n} 2^{n-i} A[\text{bit } i]$. If a is an integer then $\mathsf{num2str}_n(a)$ is the unique n-bit string A such that $\mathsf{str2num}(A) = a \bmod 2^n$. By $A + B$ we denote $\mathsf{num2str}_n(\mathsf{str2num}(A) + \mathsf{str2num}(B))$. By iA, where $i \geq 0$ is a positive integer, we mean the string $\mathsf{num2str}_n(i \cdot \mathsf{str2num}(A))$. The $\cdot$ symbol in the last expression means multiplication in the integers.)

Given a k-bit key K, derive from it a key L by way of $L = E_K(0^n) \vee 0^{n-1}1$. This ensures that L is odd.

Definition of PMAC We now define PMAC. When addition and multiplication are as just given, we are defining PMAC/add. Given a string M, its MAC is computed as illustrated in Figure 1 and as specified below.

```
Algorithm PMAC

Let m = max{1, ⌈|M|/n⌉}
Let M[1], ..., M[m] be strings s.t. M[1] ··· M[m] = M and |M[i]| = n for 1 ≤ i < m

for i = 1 to m − 1 do
        C[i] = E_K(M[i] + iL)

if |M[m]| = n then preTag = C[1] ⊕ C[2] ⊕ ··· ⊕ C[m − 1] ⊕ M[m]
                   Tag = E_K(preTag + Final(L))
              else W = pad_n(M[m])
                   preTag = C[1] ⊕ C[2] ⊕ ··· ⊕ C[m − 1] ⊕ W
                   Tag = E_K(preTag)

T = Tag[bits 1 to tagLen]
return T
```

As the MAC is deterministic, a separate MAC verification algorithm need not be given: the algorithm is to compute the MAC that should accompany the message, and see if it matches the MAC received.

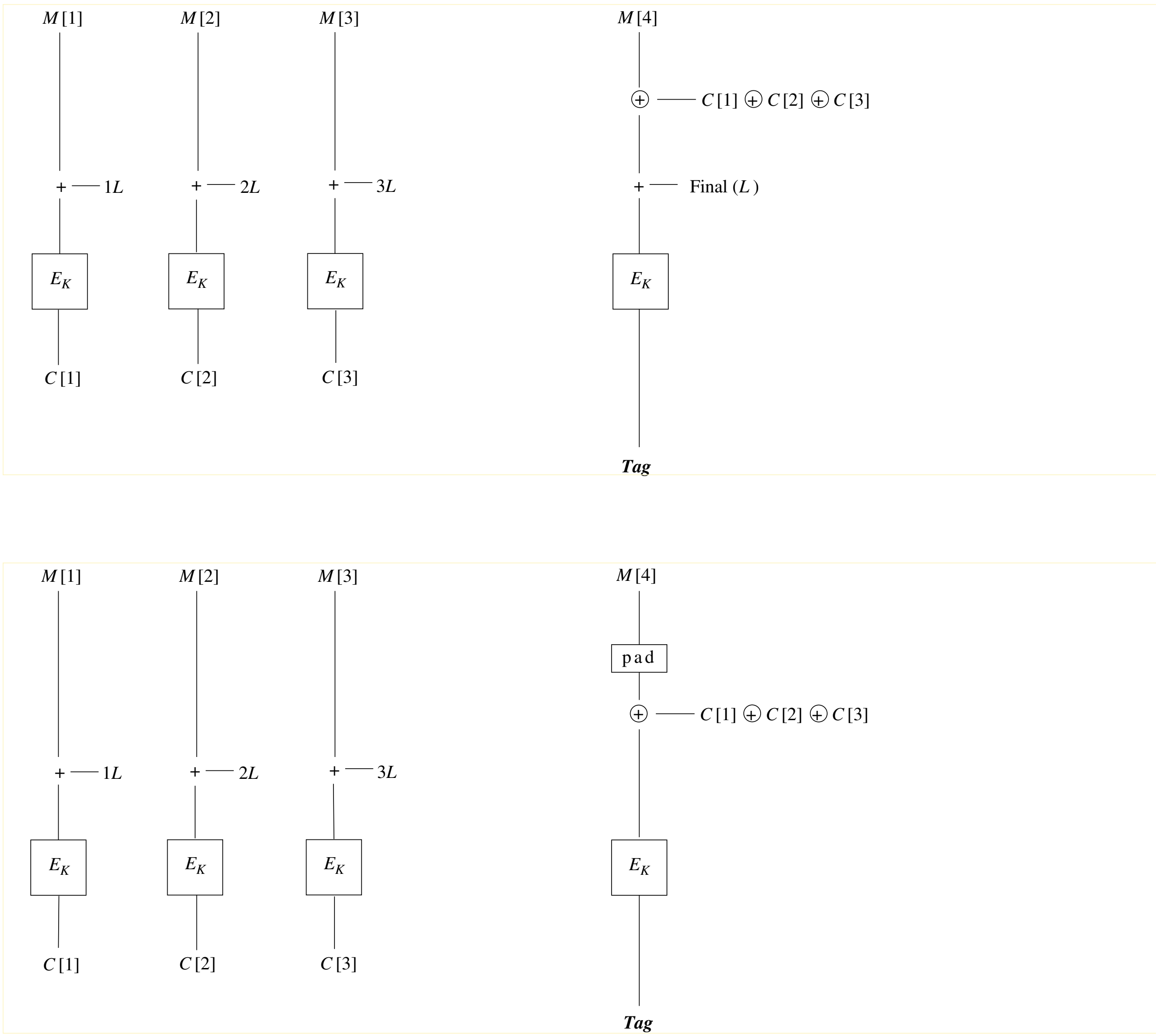

Figure 1: PMAC, *illustrated on top for a message* $M = M[1]\ M[2]\ M[3]\ M[4]$ *where each block has* n *bits, and illustrated on the bottom for a message* $M = M[1]\ M[2]\ M[3]\ M[4]$ *where the last block has fewer than* n *bits. The value* L *is determined from the underlying key* K*. The MAC is* Tag*, or a prefix of* Tag*.*

Scheme	Meaning of $A+B$	Meaning of iL, for $i \geq 1$	Meaning of Final(L)	Definition of L
PMAC/add	Add 128-bit numbers. Ignore any carry	Repeated addition (as defined in the prior column)	$\overline{L}$	$E_K(0^{128}) \vee 1$
PMAC/mod	Add 128-bit numbers mod p.	Repeated addition (as defined in the prior column)	$\overline{L}$	$E_K(0^{128})$
PMAC/xor	XOR	Multiply $\gamma(i)$ by L in GF(2^{128}), where $\gamma(i)$ is the ith word in canonical Gray-code ordering	$L(127)$, which is $2^{127} \times L$, this arithmetic in GF(2^{128}).	$E_K(0^{128}) \wedge$ Const

Figure 2: Three instantiation possiblities for PMAC. Here $A, B \in \{0,1\}^{128}$ and $i \in \{1,2,3,\ldots\}$. The underlying key is K and L is derived from K as specified in the rightmost column.

4 Comments

The algorithm is defined as using an underlying k-bit key K, but that key is mapped into (K, L), where L is an n-bit key. In a typical implementation, L would be computed only once, and saved.

The specification may make it appear as though an implementation would employ mutliplications. It would not. Successive additions of L would be used instead, as in:

$Offset = L$
for $i = 1$ to $m-1$ **do**
$\quad C[i] = E_K(M[i] \; + \; Offset)$
$\quad Offset = Offset + L$

The chain of additions used above might seem to imply that PMAC (without multiplies) is actually sequential. This again is not correct. To illustrate what goes on in a parallel implementation, suppose one has two processors, P_1 and P_2, and one wants to MAC $M = M[1] \cdots M[m]$. Start processor P_1 with $Offset = 0^n$, and start processor P_2 with $Offset = L$. Processor P_1 will be responsible for odd-indexed words while P_2 will handle even-indexed ones. Each increments its own $Offset$ by $2L$, not by L. Processor P_1 will encipher $M[1], M[3], M[5], \ldots$ and XOR the ciphertexts. Processor P_2 will encipher $M[2] \oplus M[4] \oplus M[6] \oplus \cdots$ and XOR the ciphertexts. Given the XOR'ed ciphertexts, the final MAC can then be computed by one of the processors.

Note that neither MAC generation nor MAC verification requires use of the function E^{-1}. Thus E^{-1} needn't be implemented.

Note that PMAC is incremental with respect to block substitutions [1]. In particular, if a message should change by replacing some r blocks, it takes time proportional to r to compute a revised MAC for it, assuming that one has kept around the old (n-bit) MAC.

Since $n = 128$ for us, n-bit additions are not completely trivial, especially in high-level programming languages. See Section 6 for an alternative.

Give a block cipher F, what we have called E can be either F or F^{-1}; one should let E be whichever is faster to compute.

5 PMAC/mod: Trading the Ring $\mathsf{Z}/2^n\mathsf{Z}$ for the Field Z_p

This section gives a slight variant of PMAC.

A better security bound for PMAC can be obtained by computing iL modulo p, instead of computing iL modulo 2^n. Here $p = 2^n - \delta$ is prime, for some small number δ. When addition/multiplication is defined under this revised semantics, L need not be odd; select $L = E_K(0^n)$ instead. See Figure 2.

Slightly more efficient still, we change the semantics of addition to be that one drops the carry bit but increments the sum by δ whenever a carry is generated. Multiplication by a positive number i means repeated addition.

6 PMAC/xor: A Gray-Code Trick and the Field $\mathrm{GF}(2^n)$

In this section we describe yet another method of offsetting the blocks $M[1], M[2], \ldots, M[m-1]$: we will change the semantics of $+$ to be XOR (that is, addition in GF(2^n)) and we will change the semantics of iL as well. When $\bmod 2^{128}$ additions are inconvenient, this approach may be preferred. We assume in this section that $n = 128$.

Notation If i is a positive integer then $\mathsf{ntz}(i)$ is the number of trailing 0's in the binary representation of i. So, for example, $\mathsf{ntz}(1) = \mathsf{ntz}(3) = 0$, $\mathsf{ntz}(2) = 1$, and $\mathsf{ntz}(24) = 3$. If L is an n-bit string, then $L \ll 1$ means a left shift of L by one bit (msb disappearing, and a zero coming into the lsb). Similarly, $L \gg 1$ means a right shift of L by one bit (lsb disappearing, and a zero coming into the msb).

Algorithm Given a key K for E derive from it an n-bit key L by way of $L = E_K(0^n) \wedge 0^2 1^{30} 0^2 1^{30} 0^2 1^{30} 0^2 1^{30}$. This ensures that the top two bits of every 32-bit word are zero, allowing for some pleasant implementation optimizations. Now define $L(0) = L$ and, for $i \geq 0$, define

$$L(i+1) \quad = \quad \begin{cases} L(i) \ll 1 & \text{if } \mathsf{msb}(L(i)) = 0 \\ (L(i) \ll 1) \oplus 0^{120} 1 0^4 1^3 & \text{if } \mathsf{msb}(L(i)) = 1 \end{cases}$$

Now given a string M, the PMAC algorithm proceeds as we have defined already, but with addition being defined as bitwise XOR, and iL being defined by

$$iL \quad = \quad \begin{cases} 0^n & \text{if } i = 0 \\ (i-1)L \oplus L(\mathsf{ntz}(i)) & \text{if } i \geq 1 \end{cases}$$

The value $\mathrm{Final}(L)$ is defined as $L(127)$. This can be computed directly by $(L \gg 1) \oplus L(6) \oplus L(1)$ if $\mathsf{lsb}(L) = 0$, and $(L \gg 1) \oplus L(6) \oplus L(1) \oplus L \oplus 1 0^{120} 1 0^4 1 1$ if $\mathsf{lsb}(L) = 1$. Note that each offset is obtained from the previous one by XORing it with the appropriate $L(i)$. The $L(i)$ values can be computed once, in advance, or they can be computed on the fly with the specified bit twiddling.

Explanation The following explanation assumes more mathematical background than the rest of this document. Understanding this explanation is not needed for understanding the algorithm's definition.

The algorithm just given is identical to the earlier ones except that (1) addition is done in the field GF(2^{128}); and (2) the ith offset is $\gamma(i) \times L$, where γ is a particular (convenient) permutation on $\{1, 2, 3, \ldots, 2^n - 1\}$ and $j \times L$ denotes the number j, treated as a field element, multiplied (in this field) by L. Let us elaborate.

We have constructed the $L(i)$ values in such a manner that $L(i)$ is the string that represents $2^i \times L$, where 2^i and L are regarded as points in the field GF(2^{128}) and $\times$ refers to multiplication in the field. Here we are are representing points using the irreducible polynomial $p(x) = x^{128} + x^7 + x^2 + x + 1$. A string $a_{127} \cdots a_1 a_0$ corresponds to the formal polynomial $a_{127} x^{127} + \cdots + a_1 x + a_0$.

A *Gray code* on $\{0,1\}^n$ is a permutation of $\{0,1\}^n$, say $(g_0, g_1, \ldots, g_{2^n-1})$, such that g_i and g_{i+1} differ (in the Hamming sense) by just one bit. Also, g_0 and g_{2^n-1} differ in just one bit. We implicitly make use of the Gray code $\mathcal{G}(n)$ constructed as follows: $\mathcal{G}(1) = (0, 1)$, and, for $i \geq 0$, if $\mathcal{G}(i) = (g_0, \ldots, g_{2^i-1})$ then $\mathcal{G}_{i+1} = (0g_0, 0g_1, \ldots, 0g_{2^i-2}, 0g_{2^i-1}, 1g_{2^i-1}, 1g_{2^i-2}, \ldots, 1g_1, 1g_0)$. This is easily seen to be a Gray code, and it is not hard to prove that, in this code, $g_{i+1} = g_i \oplus 1 \ll \mathsf{ntz}(i)$. Thus it is easy to compute the successive words of this code.

Moving from strings to numbers, the Gray code that we are using is $\gamma(1) = 1$, $\gamma(2) = 3$, $\gamma(3) = 2$, $\gamma(4) = 6$, $\gamma(5) = 7$, $\gamma(6) = 5$, $\gamma(7) = 4$, $\gamma(8) = 12$, and so forth. The ith offset has been defined as $iL = \gamma(i) \times L$.

Comment We defined L in a way that ensures that the top two bits of every 32-bit word are 0-bits. This means that one can change L to $2L$, or change L to $4L$, or change $4L$ to $2L$, and so forth, using either two or four shift operations (on a 64-bit machine or a 32-bit machine, respectively). This means that only one time in eight does one have to obtain a new $L(i)$ value by going to memory or doing bit twiddling; the rest of the time one shifts the current αL-value to get the $\alpha' L$ value that you want. The more zero-bits one sets aside at the beginning of each word the fewer times one has to go to memory or do bit-twiddling. But one quickly gets a diminishing return, and the security bound degrades with the number of forced zero-bits. So two or three 0-bits on the top of each word is probably a good choice.

Acknowledgments

Thanks to **Virgil Gligor**, who described his XECB MAC to me at CRYPTO '00. Thanks **John Black** (University of Nevada, Reno) for extensive email discussions. Thanks to **Mihir Bellare** (UC San Diego) for further discussions. As indicated earlier, the correctness proof for PMAC is being worked out jointly with John and Mihir.

This work was carried out while the author was at Chiang Mai University (on leave of absence from the University of California, Davis).

References

[1] M. BELLARE, S. GOLDWASSER, and O. GOLDREICH. Incremental cryptography and applications to virus protection. *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing* (STOC '95). ACM Press, pp. 45–56, 1995.

[2] M. BELLARE, R. GUÉRIN, and P. ROGAWAY. XOR MACs: New methods for message authentication using finite pseudo-random functions. *Advances in Cryptology—Crypto '95.* Lecture Notes in Computer Science, vol. 963, Springer-Verlag, pp. 15–28, 1995. Available from `www.cs.ucdavis.edu/~rogaway`

[3] M. BELLARE, J. KILIAN and P. ROGAWAY. One the security of cipher block chaining. *Advances in Cryptology—Crypto '94.* Lecture Notes in Computer Science, vol. 839, Springer-Verlag, 1994. Available from `www.cs.ucdavis.edu/~rogaway`

[4] D. BERNSTEIN. How to stretch random functions: the security of protected counter sums. *Journal of Cryptology*, vol. 12, pp. 185–192, 1999.

[5] V. GLIGOR and P. DONESCU. Fast encryption and authentication: XCBC encryption and XECB authentication Modes. Manuscript, 18 August 2000. Available from `www.eng.umd.edu/~gligor/`

[6] O. GOLDWASSER, S. MICALI, and R. RIVEST. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.

[7] US NATIONAL BUREAU OF STANDARDS. DES Modes of Operation. Federal Information Processing Standard (FIPS) Publication 81, December 1980. Available from `http://www.itl.nist.gov/div897/pubs/fip81.htm`

[8] M. WEGMAN and L. CARTER. New hash functions and their use in authentication and set equality. *J. of Computer and Systems Sciences*, vol. 22, pp 265–279, 1981.