J. Cryptology (1991) 3: 63–79

Journal of Cryptology

An Implementation for a Fast Public-Key Cryptosystem[1]

G. B. Agnew, R. C. Mullin, I. M. Onyszchuk, and S. A. Vanstone
University of Waterloo,
Waterloo, Ontario, Canada

Abstract. In this paper we examine the development of a high-speed implementation of a system to perform exponentiation in fields of the form GF(2^n). For sufficiently large n, this device has applications in public-key cryptography. The selection of representation and observations on the structure of multiplication have led to the development of an architecture which is of low complexity and high speed. A VLSI implementation has being fabricated with measured throughput for exponentiation for cryptographic purposes of approximately 300 kilobits per second.

Key words. Galois field, Normal basis, Multiplication, Exponentiation, Circuit architecture.

1. Introduction

In 1976 Diffie and Hellman [1] put forth the notion of a public-key (or asymmetric-key) cryptosystem. In this system two separate and seemingly unrelated keys perform the encryption/decryption functions. In this way the encryption function (E_i for user i) can be put in the public domain without compromising the security of the system. To create such systems, a "trap-door" is planted in what is generally accepted as a hard mathematical problem, that is, the system is constructed in such a way that without the trap-door information, an attacker is faced with solving a hard mathematical problem in order to break the system.

In the years that have followed, many public-key cryptosystems have been proposed (and many have been broken) [2]. Currently, there are two systems that are considered viable and have been implemented:

(i) systems based on the difficulty of factoring the product of two large primes (commonly known as the RSA system after the inventors [3]), and
(ii) systems based on the difficulty of finding logarithms in a finite field (commonly known as discrete exponentiation [1]).

In this study we do not discuss the strength of the two systems other than to observe that similar operations can be performed in either system and comparable security

[1] Date received: March 15, 1990. Date revised: October 30, 1990.

can be obtained [4], [5]. Instead, we look at some properties of the discrete-exponentiation system which, when combined with recent discoveries in the structure of finite fields used, provide an architecture of low complexity and high speed.

2. The Systems

2.1. *RSA*

The security of the RSA system is based on the difficulty of factoring the product of two large primes. In this system, user i forms n_i as the product of two large primes p_i and q_i. The user then forms the public portion of his key E_i and the private portion D_i using the known factorization of n_i. The operations of encryption and decryption involve exponentiation and reduction modulo n_i. For example, the encipherment of a block M under user i's public key is

$$C = M^{E_i} \mod n_i.$$

To implement such a system in hardware, a number of problems arise. First, the modulus n must be on the order of 200 decimal digits (or more than 512 bits). Since each user selects a separate modulus, the implementation must be large enough to handle every user's modulus. In the current implementations, this leads to a tradeoff between speed of computation (throughput) and complexity of the device.

A summary of the current implementations of such devices can be found in [6]. Single-chip implementations exist and operate at speeds of up to 10 kbps [7].

2.2. *Discrete Exponentiation*

A method of key exchange based on discrete exponentiation was proposed by Diffie and Hellman [1]. This system involved two users (A and B) exchanging a key over an open channel. The process is based on exponentiation and reduction modulo p, a large prime. Here, users agree upon a common modulus p and a common primitive element α.

To exchange a key, users A and B choose random elements a and b, respectively, in the range $[1, p-1]$. They can calculate and exchange the values $\alpha^a \bmod p$ and $\alpha^b \bmod p$, respectively. User A, upon receiving the calculation from user B, forms $(\alpha^b)^a \bmod p$ which forms the shared secret (user B performs a similar calculation). Discrete exponentiation can also be used to perform public-key data exchange and digital signatures as shown by ElGamal [8].

We can make the following observations about the above system:

(i) Since the system uses a common modulus p, hardware can be designed to take advantage of the fixed modulus.
(ii) While we have described the implementation in terms of integers, fields of the form $GF(2^n)$ can be used.
(iii) Alternative representations such as normal basis representation can be used.
(iv) Since the same exponent is used during the encryption/decryption process, it need not be of full Hamming weight. That is, exponentiation requires

$(d-1)$ multiplications where d is the Hamming weight of the exponent. If we limit the Hamming weight, we can improve the throughput of the system (a similar procedure has been used in implementations of the RSA system where the encryption exponent E_i is chosen to be small and decryption takes advantage of the known factorization of the modulus [9]). If this is done, care must be taken so that $\binom{n}{d}$ is large enough to prevent exhaustive attacks.

3. Normal-Basis Representation

The architecture discussed here is based on a normal-basis representation of GF(2^n). An (ordered) basis N of $K = \mathrm{GF}(2^n)$ (viewed as a vector space over GF(2)) is said to be *normal* if it is of the form $\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{n-1}}$ for some element β of K. The element β is said to be a generator of the normal basis N. It is well known (see [10]) that GF(2^n) contains such a normal basis for every $n \geq 1$. It is of interest to point out that it has been shown [11] that there exists a normal basis in GF(2^n) with the additional property that a generator of the normal basis is also a primitive element of GF(2^n), that is, a generator of the multiplicative group of the field.

For $a \in \mathrm{GF}(2^n)$, let $(a_0, a_1, \ldots, a_{n-1})$ be the coordinate vector of a relative to the normal basis N. Since $x^{2^n} = x$ for every $x \in \mathrm{GF}(2^n)$, it follows that $\beta^{2^n} = \beta$. Since the operation of squaring is a linear operator in fields of characteristic 2, then a^2 has coordinate vector $(a_{n-1}, a_0, a_1, \ldots, a_{n-2})$ so that squaring is simply a cyclic shift of the vector representation of a. In a hardware implementation, squaring an element takes one clock cycle and so is negligible.

Let

$$A = \sum_{i=0}^{n-1} a_i \beta^{2^i}$$

and

$$B = \sum_{i=0}^{n-1} b_i \beta^{2^i}$$

and let

$$C = AB = \sum_{i=0}^{n-1} c_i \beta^{2^i}.$$

Now

$$C = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j \beta^{2^i} \beta^{2^j}. \tag{1}$$

Since N is a basis, we can write

$$\beta^{2^i} \beta^{2^j} = \sum_{k=0}^{n-1} \lambda_{ij}^{(k)} \beta^{2^k},$$

where $\lambda_{ij}^{(k)} \in \mathrm{GF}(2)$. Substituting and solving for c_k yields

$$c_k = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \lambda_{ij}^{(k)} a_i b_j.$$

For convenience, we can write c_k as $c_k(A, B)$. It is easily shown that $c_k(A, B) = c_0(A^{2^{n-k}}, B^{2^{n-k}})$, thus, viewing c_k as a bilinear form in the N-coordinates of A and B, c_k is obtained from c_0 by applying a k-fold cyclic shift to the variables involved. (Note for later reference that after i cyclic shifts, coordinate position j contains $a_{(j+i) \text{ modulo } n}$.)

3.1. *Examples of Multiplication in* GF(2^5)

Let $K = \mathrm{GF}(2^5)$ be viewed as the splitting field of $f(x) = x^5 + x^2 + 1$, and suppose that $f(\alpha) = 0$. If we take $\beta = \alpha^3$, then $N = (\beta, \beta^2, \beta^4, \beta^8, \beta^{16})$ is a normal basis. Carrying out the above calculations yields

$$c_i = b_{i+4}(a_{i+4} + a_{i+3} + a_{i+1} + a_i) + b_{i+3}(a_{i+4} + a_{i+2} + a_{i+1} + a_i) + b_{i+2}(a_{i+3} + a_i) + b_{i+1}(a_{i+4} + a_{i+3}) + b_i(a_{i+4} + a_{i+3} + a_{i+2}).$$

In this case, all subscripts are added modulo 5. In expanded form

$$\begin{aligned}
c_0 &= b_4(a_4 + a_3 + a_1 + a_0) + b_3(a_4 + a_2 + a_1 + a_0) + b_2(a_3 + a_0) \\
&\quad + b_1(a_4 + a_3) + b_0(a_4 + a_3 + a_2), \\
c_1 &= b_0(a_0 + a_4 + a_2 + a_1) + b_4(a_0 + a_3 + a_2 + a_1) + b_3(a_4 + a_1) \\
&\quad + b_2(a_0 + a_4) + b_1(a_0 + a_4 + a_3), \\
c_2 &= b_1(a_1 + a_0 + a_3 + a_2) + b_0(a_1 + a_4 + a_3 + a_2) + b_4(a_0 + a_2) \\
&\quad + b_3(a_1 + a_0) + b_2(a_1 + a_0 + a_4), \\
c_3 &= b_2(a_2 + a_1 + a_4 + a_3) + b_1(a_2 + a_0 + a_4 + a_3) + b_0(a_1 + a_3) \\
&\quad + b_4(a_2 + a_1) + b_3(a_2 + a_1 + a_0), \\
c_4 &= b_3(a_3 + a_2 + a_0 + a_4) + b_2(a_3 + a_1 + a_0 + a_4) + b_1(a_2 + a_4) \\
&\quad + b_0(a_3 + a_2) + b_4(a_3 + a_2 + a_1).
\end{aligned}$$

Thus, the same logic function working on successive rotations of A and B will produce all of the components of the product vector C. Massey and Omura [12] point out a structure which takes advantage of the symmetry in calculating the terms of the product vector C. This architecture combines all of the appropriate terms of A and B in one step to form the corresponding product term c_k. Each term of C is successively generated by shifting the A and B vectors. Thus C is calculated in n clock cycles. While this structure is much simpler than a general multiplier, the structure of the connections in the logic function is prohibitively large for any n useful for cryptographic purposes.

We also note here that the choice of normal basis is important. For a random choice of normal basis, the number of nonzero terms in the bilinear form of c_0 appears to be of the order $(n^2/2)$. Let $C(N)$ denote the number of such terms in the bilinear form of c_0 relative to the normal basis N. It is shown in [16] that $C(N) \geq 2n - 1$. If equality occurs, the basis N is referred to as an *optimal* normal basis. Not every field contains an optimal normal basis, but when computing in a

field which does, there are advantages to choosing such a basis for the architecture discussed here. (The results in [16] have been generalized in [20].)

4. A Regular Architecture for Normal-Basis Multiplication

A close examination of the general nature of the above equations is necessary in order to produce an alternate architecture which is realizable for large values of n. The architecture developed in this section has the advantage that it is regular, that is, it can be implemented in VLSI technology as a linear series of more or less identical, interconnected cells.

The equations in the above example suggest that it may be convenient to write equation (1) as

$$c_k = \sum_{j=0}^{n-1} b_j \sum_{i=0}^{n-1} \lambda_{ij}^{(k)} a_i.$$

By the cyclic relation amongst the equations, this equation can be written as

$$c_k = \sum_{j=0}^{n-1} b_{j+k} \sum_{i=0}^{n-1} \lambda_{ij}^{(0)} a_{i+k},$$

where subscripts are to be reduced modulo n.

Let

$$F_j^{(k)} = b_{j+k} \sum_{i=0}^{n-1} \lambda_{ij}^{(0)} a_{i+k},$$

where again, subscripts are reduced modulo n. Note that for fixed j and variable k, the functions F are related by the cyclic permutation of subscripts. Also

$$c_k = \sum_{j=0}^{n-1} F_j^{(k)}, \qquad k = 0, 1, \ldots, n-1.$$

The functions $F_j^{(k)}$ are referred to as terms. Further, for nonnegative integers t, let

$$F_j^{(k)}(t) = b_{j+k+t} \sum_{i=0}^{n-1} \lambda_{ij}^{(0)} a_{i+k+t}.$$

A set of terms $X = \{F_0^{(k_0)}, F_1^{(k_1)}, \ldots, F_{n-1}^{(k_{n-1})}\}$ is said to be a transversal of the above set of bilinear forms if all of the k_i are distinct, $i = 0, 1, \ldots, n-1$, and if all of the residues $m_j = k_j + j$ modulo n are also distinct. (The latter condition guarantees that all of the subscripts of the b's involved are distinct.)

It is a trivial observation that

$$F_j^{(k_i)}(-k_i) = F_j^{(0)}$$

so that

$$c_0 = \sum_{i=0}^{n-1} F_i^{k_i}(-k_i).$$

Let X be a transversal $\{F_j^{(k_j)}\}_{i=0}^{n-1}$. Let $A_1, A_2, \ldots, A_n$ and $B_1, B_2, \ldots, B_n$ be cells

of cyclic shift registers **A** and **B**, respectively.[2] Define logic cells C_i, $i = 0, 1, 2, \ldots, n-1$, as follows.

In cell C_{k_j} let there be a logical circuit which will compute the expression

$$T_{k_j}(t) = \bar{B}_{j+k_j}(t) \sum_{i=0}^{n-1} \lambda_{ij}^{(0)} \bar{A}_{i+k_j}(t),$$

where $\bar{A}_p(t)$ and $\bar{B}_q(t)$ are the contents of cells A_p and B_q of **A** and **B**, respectively, at time t. This cell also contains a storage register R_k which can store previously calculated results and can add its contents $\bar{R}_k$ to the value of T_k calculated above. (Here multiplication represents the logical operation "AND" and addition represents the logical operation "XOR.") To accomplish this physically, cell C_k must be connected to the cell B_{j+k_j} of **B** and to w_j cells of **A** where w_j is the number of nonzero coefficients in $\{\lambda_{ij}^{(0)}: i = 0, 1, \ldots, n-1\}$.

Since the k_i are distinct, a cell C_i has been uniquely defined for $i = 0, 2, \ldots, n-1$. Further, recall that the m_j are also distinct, where $m_j = j + k_j$, so each of the cells of **B** is connected to precisely one of the cells of $\mathbf{C} = C_0, C_1, \ldots, C_{n-1}$. In particular, cell B_{m_j} is connected to cell $C_{m_j - j}$. Now, both $K = (k_0, k_1, \ldots, k_{n-1})$ and $M = (m_0, m_1, \ldots, m_{n-1})$ are permutations of $(0, 1, \ldots, n-1)$. Invert the permutations K and M by defining $j(i)$ to be the subscript such that $k_{j(i)} = i$ and $k(i)$ to be that subscript such that $m_{k(i)} = i$, $i = 0, 1, 2, \ldots, n-1$. In each case, cell B_i is connected to cell $C_{m_{k(i)} - k(i)}$. This is illustrated in Fig. 1(a). (For the sake of simplicity, the connections between the cells of **A** and corresponding cells of **C** are omitted.)

The network is considered to operate as follows. The system is initialized by loading the values a_i and b_j in the respective cells of A_i and B_j, respectively, and the registers R_i of cells C_i are loaded with zero for $i = 0, 1, \ldots, n-1$. At time t, for $t = 0, 1, \ldots, n-1$, the term $T_k(t)$ is calculated in cell $C_k(t)$ using the current contents of the **A** and **B** registers. The current contents of R_k are XOR'ed with $T_k(t)$ and the results are stored in register $R_{k+1 \bmod n}$ (see Fig. 1(b)).

It is claimed that at the end of time $t = n-1$, the register R_k contains c_k, $k = 0, 1, 2, \ldots, n-1$. First consider the contents of register R_0 at the end of time $n-1$. By the functioning of the network, this register contains $\sum_{t=0}^{n-1} T_t(t)$ but

$$\sum_{t=0}^{n-1} T_t(t) = \sum_{t=0}^{n-1} T_{k_{j(t)}}(t)$$

since $k_{j(k)} = t$

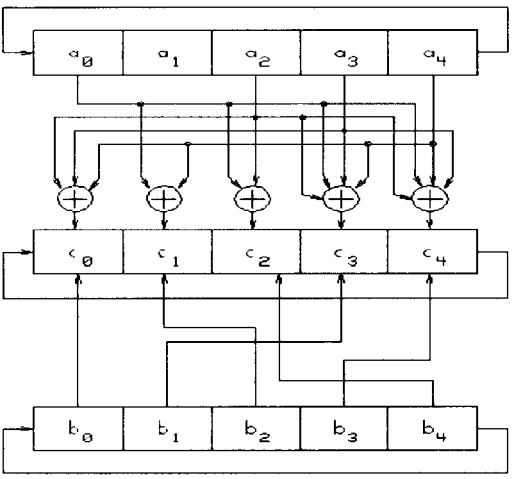
$$= \sum_{t=0}^{n-1} \bar{B}_{j(t)+k_{j(t)}}(t) \sum_{i=0}^{n-1} \lambda_{i,j(t)}^{(0)} \bar{A}_{i+k_{j(t)}}(t)$$

(and recalling that at time t, A_v and B_v contain a_{v-t} and b_{v-t}, respectively)

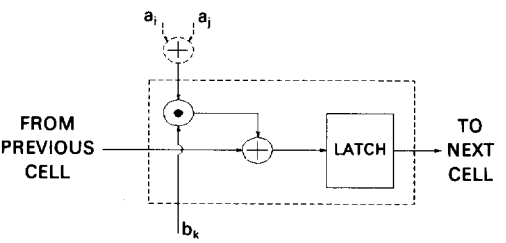
$$= \sum_{t=0}^{n-1} b_{j(t)+k_{j(t)}-t} \sum_{i=0}^{n-1} \lambda_{i,j(t)}^{(0)} a_{i+k_{j(t)}-t}$$

$$= \sum_{t=0}^{n-1} b_{j(t)} \sum_{i=0}^{n-1} \lambda_{i,j(t)}^{(0)} a_i$$

[2] An alternative derrivation of the above analysis is presented in Appendix A.



(a)



(b)

Fig. 1. (a) Register organization. (b) Cell structure.

(again since $k_{j(t)} = t$)

$$= \sum_{t=0}^{n-1} b_t \sum_{i=0}^{n-1} \lambda_{it}^{(0)} a_i = c_0.$$

Similarly, register R_s contains $\sum_{t=0}^{n-1} T_{s+t}(t)$ where $s + t$ is to be reduced modulo n. As above, this reduces to c_s.

For odd n such transversals exist. The set $X_1 = \{F_i^{(-2i)}: i = 0, 1, \ldots, n-1\}$ or $X_2 = \{F_i^{(i)}: i = 0, 1, \ldots, n-1\}$ where calculations are performed modulo n, are clearly transversals.[3] For even n, such a transversal does not exist. However, the above architecture can be extended to handle the situation $n = 2^s t$ by partitioning the registers **A** and **B** into blocks of size 2^s, and using an extension of the previous architecture (see [15]). For cryptographic purposes, the important case is that in

[3] We thank the anonymous reviewer for observing that in a flat linear layout of cells, the transversal $-j/2$, with B and C stepped in the reverse direction, produces an implementation without crossover of wires. In practice, layouts use a serpentine structure for implementing long registers and thus this may or may not have impact on the final array.

which n is odd, and particularly where n is a large prime, since GF(2^n) has no proper subfields precisely under the condition that n be prime. For this reason we consider only this case in detail.

Part of the complexity of implementing arises from the interconnection between the **A** register and the register **C** containing the cells C_i. This can be reduced by appropriate choice of the basis, using an optimal normal basis when possible (see Section 5 on fanout for details). Another way of reducing the complexity is based on the observation that the bilinear form for C_s is symmetric, that is, the term $a_i b_j$ occurs if and only if the term $b_i a_j$ appears, and there is only one self-symmetric term (namely $a_{s-1} b_{s-1}$). Thus if the logic of the above architecture is implemented to accumulate exactly one of each of the symmetric pairs $a_i b_j$ and $b_i a_j$ (say $a_i b_j$), then by interchanging the initial contents of the registers **A** and **B** and rerunning the logic in a "second pass" to accumulate the terms $b_i a_j$ to the current result, the value c_s will again occur in registers R_s, provided care is taken to suppress the term $a_{s-1} b_{s-1}$ on one of the passes (i.e., so that it appears only once). By using this two-pass method, the number of connecting lines between the **A** and the **C** registers is essentially halved.

5. Fanout

As mentioned briefly in the previous section, a major concern for the design of a hardware circuit to implement our regular architecture is the interconnection required between registers **A**, **B**, and **C**. In particular, the connections between the **A** and **C** registers have not yet been addressed. What we show in this section is that a normal basis can be selected which requires at most four connections from any **A** register cell to **C** register cells. In order to do this we need to describe several constructions for optimal normal bases.

Let **N** be a normal basis and recall from Section 3 that **C**(**N**) is the number of nonzero terms in the bilinear form for c_k. If **C**(**N**) $= 2n - 1$, the basis is called optimal. The following theorem is proven in [16].

Theorem 5.1.

(a) *If* 2 *is a primitive element in* GF($n + 1$), *then* GF(2^n) *has an optimal normal basis.*

(b) *If* 2 *is a primitive element in* GF($2n + 1$), *then* GF(2^n) *has an optimal normal basis.*

(c) *If* n *is odd and* 2 *generates the quadratic residues in* GF($2n + 1$), *then* GF(2^n) *has an optimal normal basis.*

We describe a construction for an optimal normal basis for each of the cases in the theorem.

(a) Let β be a primitive $(n + 1)$th root of unity in GF(2^n). Then

$$N = \{\beta^{2^i} : 0 \le i \le n - 1\}$$

can be shown [12] to be an optimal normal basis.

(b), (c) Let β be a primitive $(2n+1)$th root of unity in $GF(2^{2n})$. Let $\gamma = \beta + \beta^{-1}$. Then

$$N = \{\gamma^{2^i}: 0 \le i \le n-1\}$$

is an optimal normal basis in $GF(2^n)$.

An optimal normal basis given by (a) is called a type I basis and one given by either (b) or (c) is called a type II basis.

Recall from Section 4 that

$$F_j^{(k)} = b_{j+k} \sum \lambda_{ij}^{(0)} a_{i+k}.$$

Define

$$S_j^{(k)} = \{a_{i+k}: 0 \le i \le n-1, \lambda_{ij}^{(0)} = 1\}.$$

For each a_i, $0 \le i \le n-1$, let

$$n_i = |\{k_j: a_i \in S_j^{(k_j)}, 0 \le j \le n-1\}|.$$

Define the *fanout* for the transversal X to be

$$f(X) = \max\{n_i: 0 \le i \le n-1\}.$$

The fanout for a transversal X gives us an upper bound on the number of connections between any cell of the A register and the C register. Since power consumption for a VLSI device is related to the number of connections ("*fanout*") of any cell, it is important to select transversals which minimize this value. The purpose of this section is to prove that a transversal can be selected for type I and II bases which has a fanout of at most four. We only prove this result for the bases given in (b) above. Any optimal normal basis has at most two a's in any grouping of terms. To determine the $b_i a_j$ terms in the bilinear form for c_0 in the basis (b) above, it can be shown that, for $0 \le i, j \le n-1$,

$\lambda_{ij}^{(0)} = 1$ *iff* i *and* j *satisfy one of the four congruences* $2^i \pm 2^j \equiv \pm 1 \pmod{2n+1}$

(see [16]).

As an example, we can determine c_0 for $GF(2^5)$ to be

$$c_0 = b_0a_1 + b_1a_0 + b_1a_3 + b_2a_3 + b_2a_4 + b_3a_1 + b_3a_2 + b_4a_2 + b_4a_4.$$

In order to simplify notation, we only consider subscripts. To this end let

$$W = \{(i,j): b_i a_j \text{ is a term of } c_0\}.$$

Clearly, if $(i,j) \in W$, then $(j,i) \in W$. Also, $(0,1)$ always belongs to W. It is convenient to partition W into sets W_1 and W_2 such that if $(i,j) \in W_1$, then $(j,i) \in W_2$ for $i \ne j$ and $(n-1, n-1) \in W_1$. Let

$$W_1 = \{(x_0, x_1): x_0 = 0\} \cup \{(x_i, x_{i+1}): 1 \le i \le n-2, x_{i+1} \ne x_{i-1}\}$$
$$\cup \{(x_{n-1}, x_{n-1}): x_{n-1} = n-1 \ne x_{n-2}\}.$$

We first prove that

$$\{x_i: 0 \le i \le n-1\} = \{i: 0 \le i \le n-1\}. \tag{2}$$

Since 2 is a generator in $GF(2n+1)$ then for an $\alpha \in GF(2n+1)$, $\alpha \ne 0$, there exists

x, $0 \leq x \leq 2n-1$, such that $\alpha = 2^x$. We use the notation

$$x = \log_2 \alpha.$$

For our purposes it is important to observe that

$$\log_2 \alpha \equiv \log_2(-\alpha) \pmod{n}.$$

In order to prove (2), we must first prove that

$$x_i \equiv \log_2(i+1) \pmod{n}, \qquad 0 \leq i \leq n-2.$$

Clearly, $x_0 \equiv \log_2 1 \pmod{n}$ and $x_1 \equiv \log_2 2 \pmod{n}$. Assume that

$$x_i \equiv \log_2(i+1) \pmod{n}$$

for $i \leq k$. Consider x_{k+1}. Recall that x_i, x_{i+1} satisfy one of

$$2^{x_i} \pm 2^{x_{i+1}} \equiv \pm 1 \pmod{2n+1}$$

and that $x_{i+1} \not\equiv x_{i-1} \pmod{n}$.

Now

$$x_k = \log_2(k+1) + ln, \qquad \text{where} \quad l = 0 \text{ or } 1$$

and

$$\begin{aligned} 2^{x_{k+1}} &\equiv \pm 1 - 2^{ln}(1+k) \pmod{2n+1} \\ &\equiv \pm 1 - (-1)^l(1+k) \pmod{2n+1}. \end{aligned}$$

Hence

$$x_{k+1} = \log_2(\pm 1 - (-1)^l(1+k)).$$

If $l = 0$, then

$$x_{k+1} = \log_2(\pm 1 - 1 - k)$$

implying $x_{k+1} \equiv \log_2(k+2) \pmod{n}$ for otherwise $x_{k+1} \equiv x_{k-1} \pmod{n}$. If $l = 1$, then

$$\begin{aligned} x_{k+1} &= \log_2(\pm 1 + 1 + k) \\ &= \log_2(k+2) \pmod{n} \end{aligned}$$

for otherwise $x_{k+1} \equiv x_{k-1} \pmod{n}$.

We conclude that

$$x_i \equiv \log_2(i+1) \pmod{n} \qquad \text{for} \quad 0 \leq i \leq n-2.$$

Suppose $x_i = x_j$ for some i, j, $0 \leq i, j \leq n-2$. Then

$$\log_2(i+1) - \log_2(j+1) \equiv 0 \pmod{n}$$

or

$$\log_2\left(\frac{i+1}{j+1}\right) \equiv 0 \pmod{n}$$

which implies

$$\left(\frac{i+1}{j+1}\right) = 1 \quad \text{or} \quad \left(\frac{i+1}{j+1}\right) = -1.$$

In the former case $i = j$ and in the latter $i + j = 2n - 1$, which is impossible given the range of i and j. Similarly, it can be shown that $x_i \neq x_{n-1}$ for any $i, 0 \leq i \leq n - 2$. Finally, note that $2n \equiv -1 \pmod{2n + 1}$ and so $\log_2 n = n - 1$. Therefore, $x_i \equiv \log_2(i + 1), 0 \leq i \leq n - 1$, and the proof is complete.

We claim that the transversal

$$x = \{F_j^{(x_j)}: 0 \leq j \leq n - 1\}$$

has a fanout of at most four. To prove this let

$$y_i = x_i + x_{i+1}, \qquad 0 \leq i \leq n - 2,$$

and

$$y_{n-1} = 2x_{n-1},$$

where for the time being we do not reduce modulo n. Now

$$\begin{aligned} y_i &= x_i + x_{i+1} \\ &\equiv \log_2(i + 1) + \log_2(i + 2) \pmod{2n + 1} \\ &\equiv \log_2(i + 1)(i + 2). \end{aligned}$$

For what values of i and j does the relation

$$y_i = y_j, \qquad 0 \leq i \leq n - 2,$$

hold? If

$$\log_2(i + 1)(i + 2) \equiv \log_2(j + 1)(j + 2) \pmod{n},$$

then

$$(i + 1)(i + 2) \equiv (j + 1)(j + 2) \pmod{2n + 1}$$

implying that

$$(i - j)(i + j + 3) \equiv 0 \pmod{2n + 1}$$

and hence

$$i = j \quad \text{or} \quad i + j = 2n - 2.$$

Because of the range of i and j we conclude that the y_i's are distinct modulo $2n + 1$. If we reduce modulo n, then no integer in the range $[0, n - 1]$ can occur more than twice as y_i values, $0 \leq i \leq n - 2$.

Finally, if

$$y_i = y_{n-1} \qquad \text{for some } i, \qquad \leq i \leq n - 2,$$

then

$$\log_2(i + 1)(i + 2) \equiv 2(n - 1) \pmod{n}$$

or

$$(i + 1)(i + 2) \equiv 2^{2(n-1)}(-1)^l \pmod{2n + 1}$$

for some $l \in \{0, 1\}$. If $l = 0$, then we have

$$4i^2 + 12i + 9 \equiv 0 \pmod{2n + 1},$$

implying $i = n - 1$ which is again impossible. Therefore no integer in the range $[0, n - 1]$ can occur more than twice as y_i values, $0 \leq i \leq n - 1$.

Since W_2 has no repeated values, it gives rise to y_i's where no integer in the range $[0, n-1]$ occurs more than twice. It now follows that the fanout for x is at most four.

6. Limited Hamming Weight Exponents

On average, an exponentiation will require $n/2$ multiplications for a randomly chosen exponent. To increase the speed of the system and to upper bound the time taken for exponentiation, a method of limiting the Hamming weight (d) of the exponent was developed.

Let

$$Y = \alpha^K$$

represent the exponentiation process. We can represent the exponent K in its binary form as

$$K = k_0 2^0 + k_1 2^1 k_3 2^3 + \cdots + k_{n-1} 2^{n-1}, \qquad k_i \in \{0, 1\}.$$

The Hamming weight of K represents the number of coefficients $k_i = 1$.

Consider a p-bit register which will be used as a vector to map its contents into an exponent of the desired weight. This is done by dividing the register's contents into segments of size $l = \lceil \log_2 n \rceil$. The l bits are used as an index to indicate that the corresponding coefficient, $k_{f(x)}$ is 1 in the binary representation of the actual exponent (let $f(*)$ be the integer corresponding to the bit pattern). If d such segments are used, an exponent of Hamming weight of d is realized.[4] (For example, if $n = 593$, we divide the exponent register into $l = 9$ bit segments, each segment pointing to a 1 in the actual exponent.) Thus

$$K = g(K'),$$

where K' represents the contents of the exponent register and $g(K')$ represents the mapping of those contents into the actual exponent as mentioned above.

As an example, consider the case of a 20 MHz clock, $n = 1000$ bits and $d = 150$ bits in the exponent. This would require about 6.5 ms for an exponentiation or would be able to support a throughput of 150 kbps.

7. A Conventional Cryptographic System Based on Discrete Exponentiation

Once a key has been exchanged using public-key techniques, little is gained by continued use of the public-key system. In fact, a penalty is paid in terms of transmission bandwidth since $2n$ bits are required to pass n bits of information using the ElGamal scheme. To avoid this, an initial key K_0 can be passed using the public-key technique, then the message can be encrypted in the following way. Assume the message $\mathcal{M}$ is divided into n-bit blocks to form $\{M_1, M_2, \ldots, M_m\}$. To

[4] The Hamming weight is upper bounded by d. Collisions of the binary vectors can occur and reduce the actual Hamming weight of the exponent but, for typical values of d, this probability is not high.

encrypt message block M_i, the corresponding ciphertext is

$$C_i = \alpha^{K_i} \cdot M_i,$$

where

$$K_i = g(K_i')$$

and

$$K_i' = v(K_{i-1}'),$$

where $v(*)$ represents the result of a one position permutation of the contents of the exponent register. In our implementation, the various cells of the exponent register are connected using a primitive connection polynomial to form a Maximal Length Linear Feedback Shift Register (MLLFSR). In this way, a new exponent K_i is created for each block in the message.

To decrypt a block, we form

$$M_i = (\alpha^{-1})^{K_i} \cdot C_i,$$

where again, $K_i = g(K_i')$. This requires the initial calculation of α^{-1} which can then be preprogrammed into the device (see [13] for details). A number of advantages are inherent in this system: first, encryption and decryption are symmetric in terms of the number of operations (time) and thus it can be used for encryption of real-time information (note, this is not true in most public-key cryptosystems). Secondly, repeated blocks will be enciphered under different exponents thus eliminating the weakness inherent in Electronic Codebook (ECB) mode of ciphers. In addition, the individual blocks (M_i) of the message are linked by the sequence of K_i' and any addition, deletion, or reordering of blocks will be detectable.[5]

7.1. *A Technique for Generating Message Authentication Codes*

Many communication applications require authentication without secrecy. To ensure the integrity of data transmitted across a network, some means of detecting changes (either intentional or due to noise) must be included in the message. The codes used to perform this operation are Message Authentication Codes (MAC). For cryptographic purposes, the algorithm used to generate the MAC must have the following properties:

(i) The MAC-generation algorithm should be independent of the message length.
(ii) Any change such as additions, deletions, or modifications of as little as one message bit should produce an unpredictable change in about 50% of the MAC bits.
(iii) It should be computationally infeasible to generate deterministically two messages with the same MAC.

The properties listed above are satisfied by a one-way hashing function, that is, a function that forms a condensed, fixed-sized image of the message and is easy to

[5] The price paid for this is the requirement to include a method for re-establishing cryptographic synchronization in the event of loss. A protocol to handle this situation is described in [18].

calculate in the forward direction but computationally infeasible to invert. For sufficiently large fields, discrete exponentiation is a one-way function.

With this as our base, we now proceed to describe the functions which are used to combine the components of the message. As before, let K' be the vector form of the exponent (for our implementation, the exponent register will be 256 bits in length) and let $K = g(K')$ be the mapping of K' into the actual exponent (this will be 593 bits in our implementation of $GF(2^{593})$).

Again, let $f(*)$ be the integer corresponding to the bit pattern $*$. Also, let $\mathcal{M}$ be the message to be authenticated and let M_i be the ith 593-bit block of the message. In this form

$$\mathcal{M} = \{M_1, M_2, \ldots, M_m\},$$

where $\{*\}$ represents the concatenation of components. Now, K' is initially set to a fixed (secret) value (Initialization Vector—IV), K'_1. The first block of $\mathcal{M}$ is processed as

$$C_1 = M_1^{g(K'_1)}.$$

This block is then cyclically added to K' using the circuit shown in Fig. 2.[6] The result is that the qth bit of the next exponent $K'_{(i+1)}$ is calculated using the algorithm:

carryin = 0
for q from 0 to 592
  if $q < 256$
    $K_{i+1}^{(q)\prime} = (K_i^{(q)\prime} + C_i^{(q)} + carryin) \bmod 2$
    Set or Clear carryin
  end if
  if $q \geq 256$
    $j = q \bmod 256$
    $K_{i+1}^{(j)\prime} = (K_{i+1}^{(j)\prime} + C_i^{(q)} + carryin) \bmod 2$
    Set of Clear carryin
  end if

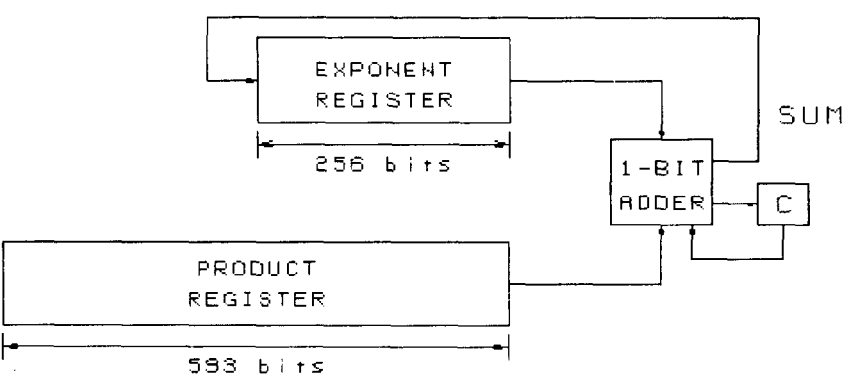


Fig. 2. MAC generation structure.

[6] The 593-bit result C_1 is added to the 256-bit exponent K'_1 in such a way that the bits of K' overlap in the calculation.

7.2. *Security of MAC-Generation Algorithm*

How well does the algorithm described in the previous section meet the requirements for an MAC generator? Two forms of operations are used in the algorithm described: discrete exponentiation and integer arithmetic.

7.2.1. *Property* 1—*Discrete Exponentiation.* The model we use is that discrete exponentiation (i.e., $Y = M^X$) produces a random mapping from M to Y in GF(2^n) for nonzero M, X and M, $X \neq 1$. Let M^* be a message block which differs in exactly one bit position from M. Let $Y^* = (M^*)^x$. Define $HW[Y^* - Y]$ to be the Hamming Weight of the difference between Y and Y^*. For our model $HW[Y^* - Y]$ is a random variable with mean $n/2$. Observations have shown that this model is appropriate for our system. Thus, any change in the message block will have the desired effect on the MAC generated within that block.

7.2.2. *Property* 1A—*Changing Exponents.* Let x^* be an exponent which differs in exactly one bit position from x. Using the previous model

$$Y = M^x,$$

$$Y^* = M^{x^*}.$$

Our experience suggests that $HW(Y - Y^*)$ will again be a random variable with mean $n/2$. We note that in using the vector form of the exponent as described in Section 6, a single bit change in the exponent representation K' will have the effect of changing two bits in the actual exponent $K = g(K')$ except in the case where this single bit change produces $g(K') = g(K'^*)$.[7]

7.2.3. *Property* 2—*Integer Addition.* In this case we examine the effect of the integer addition of the result C_i and the current exponent K_i. Integer addition was chosen to allow carries to propagate through the exponent. From our observations, the distribution of carries and their propagation appears to be a random process. Thus the linkage of the exponent used for one message block to the next will have some randomness associated with it.[8]

7.3. *Security Summary of MAC-Generation Algorithm*

It is clear that the MAC-generation algorithm has the property of linking message blocks, that is, the MAC is dependent on all blocks of the message. From the properties of the elementary operations used, a single bit change in a message block will change the resultant block in about 50% of its bit places. This result will then be added to the current exponent K'_i, and the carries introduced in the addition will randomly propagate through K'_{i+1}.

Can an attacker deterministically generate a message $\mathcal{M}^*$ which produces the same MAC?

---

[7] This can only occur when a collision occurs in the vectors and the bit change maps into another collision. For the implementation discussed here, the probability of this occurring is extremely small.

[8] A similar technique is presented in [19] by Rueppel.

7.3.1. *Case* 1—*Substitution of a Single Block.* To substitute a single block, the attacker must find M_i^* such that

$$(M_i^*)^{g(K_i')} = (M_i)^{g(K_i')}.$$

This seems at least as difficult as taking logarithms in GF(2^n).

7.3.2. *Case* 2—*Substitution of Two or More Blocks.* If the attacker could change or add two blocks (as in a meet-in-the-middle attack) such that, given K_{i-1}' and K_{i+1}', $(M_i^*)^{g(K_i')}$ followed by

$$(M_{i+1}^*)^{g(K_{i+1}^*)} \Rightarrow K_{i+2}'.$$

Again, it appears that the attacker must have the ability to take logarithms over GF(2^m).

In the above section we have shown that the MAC-generation algorithm proposed achieves the desired properties for such a system. It is observed that the MAC-generation algorithm as implemented is also computationally efficient when implemented using the architecture for the VLSI device in normal bases.

7.4. *Past and Future Development*

In 1986 a prototype of the above system was created at the University of Waterloo using custom gate arrays to implement the multiplier architecture. The device was built for $n = 593$ bits on an 11 × 17 in. board. This bit size was chosen to be realizable using gate-array technology and still provide an acceptable level of security [4]. The measured throughput of the device was approximately 300 kbps using a 15 MHz clock rate and exponents with an average Hamming weight of 150. Development continued and in 1988 a VLSI implementation was designed using 2μ technology. This device requires less than 90,000 transistors which is far less complex than other implementations of public-key devices of this bit size. As more knowledge is gained in the structure of these systems, further development of even faster, more secure public-key cryptosystems will continue.

Acknowledgment

The authors would like to thank Mike Walker for valuable discussions concerning the fanout problem of Section 5.

Appendix A

The following derivation of the multiplier structure was suggested by one of the anonymous reviewers. Let k_i, $0 \le i < n$, be some arbitrary constants.

$$u \cdot v = \sum_{t=0}^{n-1} \left(\sum_{j=0}^{n-1} b_{j+t} \sum_{i=0}^{n-1} \lambda_{ij}^{(0)} a_{i+t} \right) \alpha^{2^t}$$

$$= \sum_{j=0}^{n-1} \left[\sum_{t=0}^{n-1} b_{j+t} \left(\sum_{i=0}^{n-1} \lambda_{ij}^{(0)} a_{i+t} \right) \alpha^{2^t} \right]$$

$$= \sum_{j=0}^{n-1} \left[\sum_{t=0}^{n-1} b_{j+k_j-t} \left(\sum_{i=0}^{n-1} \lambda_{ij}^{(0)} a_{i+k_j-t} \right) \alpha^{2^{k_j-t}} \right]$$

$$= \sum_{j=0}^{n-1} \left[\sum_{t=0}^{n-1} b_{j+k_j-t} \left(\sum_{i=0}^{n-1} \lambda_{ij}^{(0)} a_{i+k_j-t} \right) \alpha^{2^{k_j}} \right]^{2^{n-t}}.$$

As in the case described in the body of the paper, we compute, for all j at position k_j, $\bar{B}_{j+k_j}(\sum_{i=0}^{n-1} \lambda_{ij}^{(0)} \bar{A}_{i+k_j})$.

References

1. Diffie, W., and M. Hellman, New directions in cryptography, *IEEE Transactions on Information Theory*, Vol. 22, 1976, pp. 472–492.
2. Diffie, W., The first ten years of public-key cryptography, *Proceeding of the IEEE*, Vol. 76, May 1988, pp. 560–577.
3. Rivest, R., A. Shamir, and L. Adleman, A method of obtaining digital signatures and public-key cryptosystems, *Communications of the ACM*, Vol. 21, pp. 120–126.
4. Blake, I., P. Van Oorschot, and S. Vanstone, Complexity issues for public-key cryptography, *Proceedings of the Nato Advance Research Institute Conference*, Ciocco, Italy, July 1986.
5. Coppersmith, D., Cryptography, *IBM Journal of Research and Development*, March 1987, pp. 244–248.
6. Beth, T., and D. Gollman, Algorithm engineering for public-key algorithms, *IEEE Journal on Selected Areas in Communication*, Vol. 7, No. 4, May 1989, pp. 458–466.
7. Brickell, E., A survey of hardware implementations of RSA, *Proceedings of Crypto* '89, Santa Barbara, CA, August, 1989.
8. ElGamal, T., A public-key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Transactions on Information Theory*, Vol. 31, 1985, pp. 469–472.
9. Hastad, J., On using RSA with low exponent in a public-key network, *Advances in Cryptography—Crypto* '85, Springer-Verlag, New York, 1986, pp. 403–408.
10. Ore, O., On a special class of polynomials, *Transactions of the American Mathematical Society*, Vol. 35, 1933, pp. 559–584.
11. Lenstra, H. W., and R. J. Schoof, Primitive normal bases for finite fields, *Mathematics of Computation*, Vol. 48, 1987, pp. 217–232.
12. Wah, P., and M. Wang, Realization and application of the Massey–Omura lock, *Proceedings of the 1984 International Zurich Seminar on Digital Communications*, March 1984, pp. 175–182.
13. Agnew, G., R. Mullin, and S. Vanstone, Arithmetic operations in GF(2^n), Submitted to the *Journal of Cryptology*.
14. Omura, J., and J. Massey, U.S. patent #4,587,627, May, 1986.
15. Onyszchuk, I., R. C. Mullin, and S. A. Vanstone, U.S. patent #4,745,568, May 1988.
16. Mullin, R. C., I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson, Optimal normal bases in GF(p^n). *Discrete Applied Mathematics*, Vol. 22, 1988–89, pp. 149–161.
17. Rosati, T., A high-speed data encryption processor for public key cryptography, *Proceeding of IEEE Custom Integrated Circuits Conference*, San Diego, CA, May 1989, pp. 12.3.1–12.3.5.
18. Agnew, G., R. Mullin, and S. Vanstone, An interactive data exchange protocol based on discrete exponentiation, *Proceedings of Eurocrypt* '88, May 1988, Lecture Notes in Computer Science, Vol. 330, Springer-Verlag, Berlin, pp. 159–166.
19. Rueppel, R., Correlation immunity and the summation generator, *Proceedings of Crypto* '85, Lecture Notes in Computer Science, Vol. 218, Springer-Verlag, Berlin, pp. 260–272.
20. Ash, D., I. Blake, and S. Vanstone, Low complexity normal bases, *Discrete Applied Mathematics*, Vol. 25, 1989, pp. 191–210.