An Artificial Immune System Heuristic for Generating Short Addition Chains

Nareli Cruz-Cortés, Francisco Rodríguez-Henríquez, *Member, IEEE*, and Carlos A. Coello Coello, *Senior Member, IEEE*

***Abstract*—This paper deals with the optimal computation of finite field exponentiation, which is a well-studied problem with many important applications in the areas of error-correcting codes and cryptography. It has been shown that the optimal computation of finite field exponentiation is a problem which is closely related to finding a suitable addition chain with the shortest possible length. However, it is also known that obtaining the shortest addition chain for a given arbitrary exponent is an NP-hard problem. As a consequence, heuristics are an obvious choice to compute field exponentiation with a semi-optimal number of underlying arithmetic operations. In this paper, we propose the use of an artificial immune system to tackle this problem. Particularly, we study the problem of finding both the shortest addition chains for exponents e with moderate size (i.e., with a length of less than 20 bits), and for the huge exponents typically adopted in cryptographic applications, (i.e., in the range from 128 to 2048 bits).**

***Index Terms*—Artificial immune systems (AIS), cryptography, heuristics, shortest addition chains.**

I. Introduction

FIELD OR MODULAR exponentiation is heavily utilized in several major public-key cryptosystems such as RSA, Diffie–Hellman and DSA [4], [24]. For instance, the RSA encryption/decryption scheme is based on the computation of $M^e \bmod n$, where e is a fixed number, M is an arbitrarily chosen numeric message and $n = pq$ is the product of two large primes p, q. Additionally, modular exponentiation is also used in computational number theory including applications on integer prime testing, integer factorization, field multiplicative inverse computation, etc.

A finite field or Galois field (so named after Evariste Galois) is a set having finitely many elements in which the usual arithmetic operations (addition, subtraction, multiplication, division by nonzero elements) are well defined. Moreover, all usual algebraic laws, namely, commutative, associative and distributive laws, hold [24]. The order of a finite field is defined as the number of elements q that it contains. Typical modern cryptographic applications utilize finite fields with a size q of as much as 2^{1024} or more field elements [32].

Manuscript received April 22, 2005; revised October 15, 2005 and January 9, 2006. The work of N. Cruz-Cortés was supported by Project SIP-IPN 20072170. The work of F. Rodríguez-Henríquez was supported in part by CONACyT under Project 45306-Y. The work of C.A. Coello Coello was supported in part by CONACyT under Project 45683-Y.

N. Cruz-Cortés is with the Center for Computing Research, National Polytechnic Institute, México (e-mail: nareli@cic.ipn.mx).

F. Rodríguez-Henríquez and C. A. Coello Coello are with CINVESTAV-IPN, Departamento de Computación, México (e-mail: francisco@cs.cinvestav.mx; ccoello@cs.cinvestav.mx)

Digital Object Identifier 10.1109/TEVC.2007.906082

If $q = p$, with p a prime, then the set of integers modulo p, form a *prime finite field*, denoted as $F = \mathrm{GF}(p)$. In a prime finite field, any arbitrary element $A \in F$ is simply an integer in the range $A \in \{0, 1, 2, \cdots, p-1\}$. In order to guarantee that any arithmetic operation within this field will result in an integer within that range, operations are computed by taking the remainder on integer division by p. As a simple example of a prime finite field consider $\mathrm{GF}(p = 17)$. That field has a total of 17 elements corresponding to the integers in the range [0, 16]. For instance, given the field elements $a = 4$ and $b = 15$, their addition $c = a + b$ and multiplication $d = a \cdot b$ can be computed as $c = a + b = 4 + 15 \bmod 17 = 2$ and $d = a \cdot b = 4 \cdot 15 \bmod 17 = 9$, respectively.

On the other hand, by setting $q = 2^n$ with n a positive integer, a *binary finite field* denoted as $\mathrm{GF}(2^n)$ is obtained. A binary finite field can be constructed by finding a monic irreducible polynomial $P(x) = x^n + p_{n-1}x^{n-1} + \cdots + p_2x^2 + p_1x + 1$ of degree n with coefficients $p_i \in [0, 1]$ for $i = 1, 2, \cdots, n-1$. The $q = 2^n$ elements of a binary finite field are the set of all polynomials with degree $n-1$ such that, $A(x) = a_{n-1}x^{n-1} + \cdots + a_2x^2 + a_1x + a_0$ with coefficients $a_i \in [0, 1]$ for $i = 0, 2, \cdots, n-1$. In an analog way to prime finite fields, all arithmetic operations are computed by taking the remainder on polynomial division by $P(x)$. As a simple example consider the binary finite field $\mathrm{GF}(2^3)$ constructed using the irreducible polynomial $P(x) = x^3 + x + 1$. Then, the $q = 2^3 = 8$ field elements are $\{0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1\}$. For instance, given the field elements $A(x) = x^2 + x$ and $B(x) = x + 1$, their addition $C = A + B$ and multiplication $D = A \cdot B$ can be computed as $C = A + B = (x^2 + x) + (x + 1) \bmod x^3 + x + 1 = x^2 + 1$ and $D = A \cdot B = (x^2 + x) \cdot (x + 1) \bmod x^3 + x + 1 = 1$.

Since both prime and binary finite fields form a group with respect to the addition and multiplication operations, the result of adding or multiplying any two arbitrary field elements will always be an element in the field.

Field exponentiation can be defined in terms of field multiplication as follows. Let A be an arbitrary element of a finite field $F = \mathrm{GF}(q)$. Also, let e be defined as an arbitrary positive integer. Then, field exponentiation of an element A raised to the power e is defined as the problem of finding an element $B \in F$ such that

$$B = A^e \bmod P \tag{1}$$

where P is either a large prime (in the case of prime finite fields) or an irreducible polynomial (in the case of binary finite fields).

Taking advantage of the linearity property of the modular operation, (1) can be evaluated by performing a reduction modulo

P at each step of the exponentiation, thus guaranteeing that all the partial results will not grow larger than twice the length of the modulus P. In the rest of this paper, we will consider that every multiplication operation always includes a subsequent reduction step.

In general, one can follow two strategies in order to optimize the computation of (1). One approach is to implement field multiplication, the main building block required for field exponentiation, as efficiently as possible. The other is to reduce the total number of multiplications needed to compute (1). In this paper, we address the latter approach, assuming that arbitrary choices of the base element A are allowed but considering that the exponent e has been previously fixed.

A large number of field exponentiation algorithms have been reported. Known strategies include: binary, m-ary, adaptive m-ary, power tree, and the factor method, to mention just a few [5]–[7], [9], [20], [26]–[30], [32], [37], [40]. Those algorithms all have in common the fact that they strive to keep the number of required field multiplications as low as possible through the usage of a particular heuristic. However, none of those strategies can be considered to yield an optimal solution for every possible field size. Obviously, the larger the size of the field utilized the harder the problem of optimizing the computation of the field exponentiation.

On the other hand, all the aforementioned methods can be mathematically rephrased by using the concept of *addition chains*. Indeed, taking advantage of the fact that the exponents are additive, the problem of computing powers of the base element A, can be directly translated to an addition calculation. The concept of an *addition chain* for a given exponent e can be informally defined as follows.

An *addition chain* for e of length l is a sequence U of positive integers, $u_0 = 1, u_1 \ldots, u_l = e$ such that for each $i > 1$, $u_i = u_j + u_k$ for some j and k with $0 \le j \le k < i$.

An addition chain dictates the correct sequence of multiplications required for performing an exponentiation. Hence, if U is an addition chain that computes e as mentioned above, then for any $A \in F$, we can find $B = A^e$ by successively computing: $A, A^{u_1}, \ldots, A^{u_{i-1}}, A^e$.

For instance, the addition chain (1,2,3,5,10,20,23,46,47) leads to the following scheme for the computation of A^{47}

$$\begin{array}{lll} A^1; & A^2 = A^1A^1; & A^3 = A^2A^1; \\ A^5 = A^3A^2; & A^{10} = A^5A^5; & A^{20} = A^{10}A^{10}; \\ A^{23} = A^{20}A^3; & A^{46} = A^{23}A^{23}; & A^{47} = A^{46}A^1. \end{array}$$

An *addition sequence* is a generalization of an addition chain where not just one but several positive integers $e_0 < e_1 \ldots < e_s$ must be included in the given sequence.

Let $l(e)$ be the shortest length of any valid addition chain for a given positive integer e. Then, the theoretical minimum number of field multiplications required for computing the field exponentiation of (1) is precisely $l(e)$. Unfortunately, the problem of determining an addition chain for e with the shortest length $l(e)$ is an **NP**-hard problem [32]. Therefore, we have no option but to use some kind of heuristic strategy in order to find an optimal addition chain when dealing with sufficiently large exponents e.

Generally speaking, a heuristic strategy tries to find in a reasonable time near-optimal results for hard optimization problems, i.e., those problems having huge search spaces. A heuristic method offers no guarantee on the quality of the solutions (if any) to be found. However, it can operate under nearly every possible set of restrictions. Typically, a heuristic method starts from a nonoptimal solution population and iteration after iteration improves its findings until a reasonable and/or valid solution can be achieved. The gradual improvement on the partial results is done using either deterministic or probabilistic search criteria. Given a fixed set of initial conditions, the optimized solutions obtained by a deterministic heuristic will remain unchanged from run to run. On the contrary, repeated executions of a probabilistic heuristic may produce different final solutions.

There has been an enormous amount of literature reporting deterministic heuristics methods for finding short addition chains on large exponents. Some examples are the aforementioned binary algorithm and its generalization, the window method, the run-length and hybrid method, and so on [20], [26], [27], [30], [37], [38].

On the other hand, relatively few probabilistic heuristics have been reported so far for finding near-optimal addition chains [6], [9], [33]. In [33], a genetic algorithm search engine was proposed for solving this optimization problem but the authors' strategy was only tested for exponents that are too small (9 bits or less) to be considered practical in serious applications. In [9], the use of an artificial immune system (AIS) was proposed as a probabilistic heuristic for finding minimal-length addition chains. Those optimal addition chains were then used for computing multiplicative inverses on binary extension fields.

In [6], an algorithm for obtaining short addition chains on 512-bit exponents was presented. That algorithm was divided into two parts: In the first phase, the computation of an addition chain for a large exponent e was reduced to the computation of an addition sequence composed by a set of integers (called windows), that are significantly smaller than e. Then, in the second phase, an addition sequence for those windows is produced. Four different search criteria were used in order to minimize the length of the addition sequences so produced. Although authors in [6] reported good experimental results, the exact method of deciding which search criterion should be used was left open (in fact, the authors mentioned that they unsuccessfully tried the simulated annealing technique).

In this paper, we propose the usage of a probabilistic heuristic based on an AIS search engine for finding short addition chains when dealing with very large exponents. We discuss the rationale behind the algorithm presented, and we compare its performance against well-known deterministic strategies using relatively small exponents, i.e., exponents with bit length m less than 12 bits. Since for those small exponents exact optimal addition chains are known (obtained by means of exhaustive search), we can find out the precise quality of the solutions obtained by our approach. Furthermore, we present a detailed description of how our proposed strategy can be extended for larger exponents e (up to 30 bits) and for very large exponents with bit length m well in the range of cryptographic applications, i.e., $m \in [128, 256, 512, 768, 1024]$ bits.

In the case of large exponents, we incorporate our AIS strategy to both phases of the algorithm presented in [6]. First, the combination of the sliding window strategy together with an AIS heuristic is utilized for efficiently partitioning a given large exponent into smaller windows. Afterwards, an AIS search engine is utilized for grouping the obtained windows into a single addition sequence. Although, in general, optimal solutions on this range are unknown, we provide a comparison of our experimental results against the ones reported by known deterministic approaches.

The rest of this paper is organized as follows. In Section II, we present a brief review of several relevant deterministic heuristics proposed in the specialized literature for computing field exponentiation. Section III describes the framework of the probabilistic heuristic approach presented in this paper, which is based on the concepts of window partitioning and addition sequences. In Section IV, the proposed AIS heuristic together with its problem representation is explained. In Section V, we describe the proposed algorithm including two design examples; one for a small exponent and another for a large 128-bit exponent. Section VI presents several experiments and statistical tests performed on the proposed AIS heuristic method. In Section VII, two code-theory applications of the AIS method are described. Finally, in Section VIII, some concluding remarks and possible paths for future research are drawn.

II. Deterministic Heuristics for Field Exponentiation

In this section, we include a brief review of the main deterministic heuristic proposed in the literature for computing field exponentiation.

A. Binary Strategies

Let e be an arbitrary m-bit positive integer e, with a binary expansion representation given as, $e = (1e_{m-2}\dots e_1e_0)_2 = 2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i$. Then

$$\mathbf{y} = \mathbf{x}^e = \mathbf{x}^{2^{m-1}+\sum_{i=0}^{m-2} 2^i e_i} = \mathbf{x}^{2^{m-1}} \cdot \prod_{i=0}^{m-2} \mathbf{x}^{2^i e_i}. \quad (2)$$

Binary strategies evaluate (2) by scanning the bits of the exponent e one by one, either from left to right (MSB-first binary algorithm) or from right to left (LSB-first binary algorithm) applying the so-called Horner's rule.[1] Both strategies require a total of $m-1$ iterations. At each iteration, a squaring operation is performed, and if the value of the scanned bit is one, a subsequent field multiplication is performed. Therefore, the binary strategy requires a total of $m-1$ squarings and $H(e)-1$ field multiplications, where $H(e)$ is the Hamming weight of the binary representation of e. The pseudocode of the MSB-first and the LSB-first binary algorithms are shown in Figs. 1 and 2, respectively. The computational complexity of the algorithm in Fig. 1 is given as

```
Input:  x, n, $e = (e_{m-1} \dots e_1 e_0)_2$
Output: $\mathbf{y} = \mathbf{x}^e \bmod n$

1. $\mathbf{y} = x$ ;
2. for $i = m-2$ downto $0$ do {
3.     $\mathbf{y} = \mathbf{y}^2$ ;
4.     if $e_i == 1$ then $\mathbf{y} = \mathbf{y} \cdot \mathbf{x}$ ; }
5. output y
```

Fig. 1. MSB—first binary exponentiation.

```
Input:  x, n, $e = (e_{m-1} \dots e_1 e_0)_2$
Output: $\mathbf{y} = \mathbf{x}^e \bmod n$

1. $\mathbf{p} = \mathbf{x}$ ; $\mathbf{y} = 1$ ;
2. for $i = 0$ to $m-1$ do
3. {  if $e_i == 1$ then $\mathbf{y} = \mathbf{y} \cdot \mathbf{p}$ ;
4.     $\mathbf{p} = \mathbf{p}^2$ } ;
5. output y
```

Fig. 2. LSB—first binary exponentiation.

$$P(e,m) = m + H(e) - 2 = \lfloor \log_2(e) \rfloor + H(e) - 1. \quad (3)$$

An Example: Let us define $e = 1903 = (11101101111)_2$. Then, $m = 11$ and $H(e) = 9$. According to (3). the computational complexity of the binary algorithm is given as

$$P(e) = m + H(e) - 2 = 11 + 9 - 2 = 18.$$

After evaluating the algorithm of Fig. 1, the resulting binary sequence is given as

$$x^1 \to x^2 \to x^3 \to x^6 \to x^7 \to x^{14} \to x^{28} \to x^{29} \to x^{58} \to x^{59} \to x^{118} \to x^{236} \to x^{237} \to x^{474} \to x^{475} \to x^{950} \to x^{951} \to x^{1902} \to x^{1903}.$$

B. Window Strategies

The binary method discussed in the preceding section can be generalized by scanning more than one bit at a time. Hence, the window method (first described in [26]) scans k bits at a time. The window method is based on a k-ary expansion of the exponent, where the bits of the exponent e are divided into k-bit words or digits. The resulting words of e are then scanned performing k consecutive squarings and a subsequent multiplication as needed. In the following, we describe the window method in a more formal way.

[1]Horner's rule, named after W. G. Horner, ranks among the most efficient algorithms for the computation of nth degree polynomials of the form, $p(x) = p_n x^n + p_{n-1}x^n - 1 + \cdots + p_1 x + u_0, p_n \neq 0$, for fixed values of x.

Horner's rule solves this problem by evaluating $p(x)$ as, $p(x) = (\dots(p_n x + p_{n-1})x + \cdots)x + p_0$.

This elegant algorithm was discovered independently by Isaac Newton 150 years earlier than Horner and by the Chinese mathematician C. C. Chao in the 13th century [26].

```
Input:  $\mathbf{x}, \mathbf{n}, e = (e_{m-1} \dots e_1 e_0)_2$, k divi-
        sor of $m$ such that $\Psi = m/k$.
Output: $\mathbf{y} = \mathbf{x}^e \bmod n$.

  1. Pre-compute and store $x^j$ for all $j = 1, 2, 3, 4, \dots, 2^k - 1$.
  2. Divide $e$ into $k$-bit words $W_i$ for $i = 0, 1, 2, \dots, \Psi - 1$.
  3. $\mathbf{y} = \mathbf{x}^{W_{\Psi-1}}$;
  4. for $i = \Psi - 2$ downto 0 do {
  5.     $\mathbf{y} = \mathbf{y}^{2^k}$ ;
  6.     if $W_i \neq 0$ then $\mathbf{y} = \mathbf{y} \cdot \mathbf{x}^{W_i}$;
     }
  7. output y
```

Fig. 3. MSB—first 2^k-ary exponentiation.

Let e be an arbitrary m-bit positive integer e, with a binary expansion representation given as

$$e = (1e_{m-2} \dots e_1 e_0)_2 = 2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i.$$

Let k be a small divisor of m. Then, this binary expansion of e can be partitioned into Ψ words of length k, such that $k\Psi = m$. If k does not divide m, then the exponent must be padded with at most $k-1$ zeros. Let us define $W_i \in [[0, 2^k - 1]]$ as

$$W_i = \left(e_{ik+(k-1)} e_{ik+(k-2)} \dots e_{ik+1} e_{ik}\right)_2 = \sum_{j=0}^{k-1} 2^j e_{(ik+j)}. \tag{4}$$

Then, we can equivalently represent e as, $\sum_{i=0}^{\Psi-1} W_i \cdot 2^{id}$. Using the above definition, we have

$$\mathbf{y} = \mathbf{x}^e = \mathbf{x}^{\sum_{i=0}^{\Psi-1} 2^{id} W_i} = \prod_{i=0}^{\Psi-1} \mathbf{x}^{2^{id} W_i}. \tag{5}$$

Equation (5) is the basis of the window MSB-first procedure for exponentiation described in the pseudocode of Fig. 3. The window method first precomputes the values of x^j for $j = 1, 2, 3, \dots, 2^k - 1$. Then, the exponent e is scanned k bits at a time from the most significant word $(W_{\Psi-1})$ to the least significant word (W_0). At each iteration, the current partial result y is raised to the 2^k power and multiplied with x^{W_i}, where W_i is the current nonzero word being processed. Referring to Fig. 3, it can be seen that:

- The first part of the algorithm consists of the precomputation of the first 2^k powers of $\mathbf{x}$ at a cost of $2^k - 2$ preprocessing multiplications.
- At each iteration of the main loop, the power $\mathbf{y}^{2^k}$ can be computed by performing k consecutive squarings. The total number of squarings is given by $(\Psi - 1)k = m - k$.
- At each iteration, one multiplication is performed whenever the ith word W_i is different than zero. Since all but one of the 2^k different values of W_i are nonzero, the average number of required multiplications is given as

$$(\Psi - 1)(1 - 2^{-k}) = \left(\frac{m}{k} - 1\right)(1 - 2^{-k}).$$

Thus, the average number of multiplications needed by the window method in order to compute an m-bit field exponentiation is given as

$$P(m, k) = (2^k - 2) + (m - k) + \left(\frac{m}{k} - 1\right)(1 - 2^{-k}). \tag{6}$$

For $k = 1, 2, 3, 4$, the window method sketched at Fig. 3 is called, respectively, *binary*, *quaternary*, *octary*, and *hexa* MSB-first exponentiation method. In particular, note that by evaluating (6) for $k = 1$, the average number of multiplications for the binary algorithm can be found as $(3/2)(m-1)$ field operations on average.

One obvious improvement of the strategy just outlined is that instead of calculating and storing all the 2^k first powers of x, one can just precompute the windows needed for a given exponent e, thus saving some operations. This last idea is illustrated in the examples below.

Example: Once again, let us consider the exponent $e = 1903 = (11101101111)_2$ with $m = 11$. Then, the window method computational complexity and resulting sequence using $k = 2, 3, 4$ can be found as follows.

Quaternary: $e = 1903 = (\mathbf{0}1\ 11\ 01\ 10\ 11\ 11)_2$
$P(m, k) = 2$ Precomp mults $+ 10$ Sqrs$+5$ mults $= 17$.
Precomp. sequence: $x^1 \to x^2 \to x^3$.
Main sequence:

$$x^1 \to x^2 \to x^4 \to x^7 \to x^{14} \to x^{28} \to x^{29} \to x^{58} \to x^{116} \to x^{118} \to x^{236} \to x^{472} \to x^{475} \to x^{950} \to x^{1900} \to x^{1903}.$$

Octal: $e = 1903 = (\mathbf{0}11\ 101\ 101\ 111)_2$
$P(m, k) = 4$ Precomp mults $+ 9$ Sqrs $+ 3$ mults $= 16$.
Precomp. sequence: $x^1 \to x^2 \to x^3 \to x^5 \to x^7$.
Main sequence:

$$x^3 \to x^6 \to x^{12} \to x^{24} \to x^{29} \to x^{58} \to x^{116} \to x^{232} \to x^{237} \to x^{474} \to x^{948} \to x^{1896} \to x^{1903}.$$

Hexa: $e = 1903 = (\mathbf{0}111\ 0110\ 1111)_2$
$P(m, k) = 6$ Precomp mults $+ 8$ Sqrs $+ 2$ mults $= 16$.
Precomp. sequence: $x^1 \to x^2 \to x^3 \to x^6 \to x^7 \to x^{14} \to x^{15}$.
Main sequence:

$$x^7 \to x^{14} \to x^{28} \to x^{56} \to x^{112} \to x^{118} \to x^{236} \to x^{472} \to x^{944} \to x^{1888} \to x^{1903}.$$

However, none of the above deterministic methods is able to find the shortest addition chain for $e = 1903$. In Section V-A, we will retake this example showing that the exponentiation for

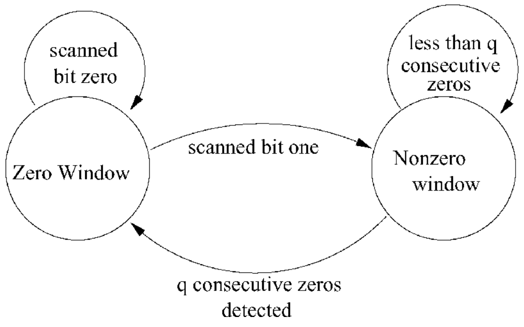

Fig. 4. Partitioning algorithm.

this example can be done using a sequence consisting of only 15 multiplication steps.

C. Adaptive Window Strategy

The adaptive or sliding window strategy is quite useful for exponentiations with extremely large exponents (i.e., exponents with bit length greater than 128 bits) mainly because of its ability to adjust its method of computation according to the specific form of the exponent at hand. This adjustment is done by partitioning the input exponent into a series of variable-length zero and nonzero words called *windows*. As opposed to the traditional window method discussed in the previous section, the sliding window algorithm provides a performance tradeoff in the sense that allows the processing of variable-length zero and nonzero digits. The main goal pursued by this strategy is to try to maximize the number and length of zero words, while using relatively large values of k.

A sliding window exponentiation algorithm is typically divided into two phases: exponent partitioning and the field exponentiation computation itself. In the first phase, the exponent e is decomposed into zero and nonzero words (*windows*) W_i of length $L(W_i)$ by using some partitioning strategy. Although, in general, it is not required that the window's lengths $L(W_i)$ must all be equal, all nonzero windows should have a length $L(W_i)$ smaller than a given number k. Let Z be the number of zero windows and NZ be the number of nonzero windows, so that their addition Ψ represents the total number of windows generated by the partitioning phase, i.e.,

$$\Psi = Z + NZ. \tag{7}$$

It is useful to force the least significant bit of a nonzero window W_i to be equal to 1. In this way, when comparing with the standard window method discussed in the previous section, the number of preprocessing multiplications are at least nearly halved, since x^w must only be precomputed for w odd.

Several sliding window partitioning approaches have been proposed [6], [7], [20], [26], [27], [30]. Proposed techniques differ in whether the length of a nonzero window has to have a constant or a variable length. The partitioning algorithm instrumented in this work scans the exponent from the most significant to the least significant bit according to the finite-state machine shown in Fig. 4. Hence, at any moment, the algorithm is either completing a zero window or a nonzero window. Zero windows are allowed to have an arbitrary length. However, the maximum length of any given nonzero window should not exceed the value of k bits.

```
Input:  $\mathbf{x}, \mathbf{n}, e = (e_{m-1} \dots e_1 e_0)_2$
Output: $\mathbf{y} = \mathbf{x}^e \bmod n$.

  1. Pre-compute and store $x^j$ for at most all $j = 1, 3, 5, \dots, 2^k - 1$.
  2. Divide $e$ into zero and nonzero windows $W_i$ of length $L(W_i)$
     for $i = 0, 1, 2, \dots, \Psi - 1$.
  3. $\mathbf{y} = \mathbf{x}^{W_{\Psi-1}}$;
  4. for $i = \Psi - 2$ downto $0$ do {
  5.     $\mathbf{y} = \mathbf{y}^{2^{L(W_i)}}$;
  6.     if $W_i \neq 0$ then $\mathbf{y} = \mathbf{y} \cdot \mathbf{x}^{W_i}$;
     }
  7. output y
```

Fig. 5. Sliding window exponentiation.

Starting from the zero window state (ZWS), the exponent bits are checked one by one. As long as the value of the current scanned bit is zero, the algorithm stays in ZWS accumulating as many consecutive zeros as possible. If the incoming bit is one, the finite-state machine switches to the nonzero window state (NZWS). The automaton will stay there as long as q consecutive zeros had not been collected. If this condition occurs, the automaton switches to ZWS (usually, q is chosen to be a small number, namely, $q \in [2, 5]$). Otherwise, if k bits can been collected, the partitioning algorithm stores the new formed nonzero window and stays in NZWS in order to generate another NZ window.

The pseudocode for the sliding window exponentiation algorithm is shown in Fig. 5. From that figure, it can be seen that:

- The first part of the algorithm consists on the precomputation of at most the first 2^k odd powers of $\mathbf{x}$ at a cost of no more than $2^{k-1} - 1$ preprocessing multiplications.
- At step 2, the exponent e is partitioned using the strategy described above and depicted in Fig. 4. As a consequence, a total of Z zero windows and NZ windows will be produced.
- At step 3, $\mathbf{y}$ is initialized using the value of the most significant window (MSW) as $\mathbf{y} = x^{W_{\Psi-1}}$. It is always assumed that $W_{\Psi-1} \neq 0$.
- At each iteration of the main loop, the power $\mathbf{y}^{2^{L(W_i)}}$ can be computed by performing $L(W_i)$ consecutive squarings. The total number of squarings is given by $m - L(W_{\Psi-1})$.
- At each iteration, one multiplication is performed whenever the ith word W_i is different than zero. Recall that NZ represents the number of nonzero windows. Therefore, the number of multiplications required at this step of this algorithm is $NZ - 1$. Although the exact value of NZ will depend on the partitioning strategy instrumented, our experiments show that an approximate value for NZ using $q = 2$, $k = 5$, is about 0.15 m.

Thus, we find that the average number of multiplications needed to compute a field exponentiation for an m-bit exponent e is given as

$$\begin{aligned} P(m,k) &= (2^{k-1}-1) + (m - L(W_{k-1})) + NZ - 1 \\ &\approx 2^{k-1} - 1 + 1.15m - L(W_{k-1}). \end{aligned} \tag{8}$$

Due to the considerable high efficiency of the partitioning strategy for collecting zero words, the sliding window method significantly outperforms the standard window method when sufficiently large exponents are computed [27]. However, notice that the value of the parameter k cannot be too large due to the exponentially increasing cost of precomputing the first 2^k odd powers of $\mathbf{x}$ (step 1 of Fig. 5). In practice and depending on the value of m, $k \in [4, 8]$ is generally adopted.

III. Addition Sequence Heuristic for Field Exponentiation

As previously mentioned, the major drawback of the sliding window method outlined in the last section is the high computational cost of increasing the value of k. This difficulty can be alleviated by using the concept of addition sequences, which is the main subject to be addressed in this section. We first give formal definitions and theoretical bounds for addition sequences. Then, we discuss how to produce short addition sequences. Finally, we introduce the sliding window method using addition sequences which is the technique adopted in this work for large exponents.

A. Mathematical Definitions

Definition: Let e be an arbitrary positive integer whose binary expansion is given as $e = e_{m-1}e_{m-2}\dots e_1e_0$, where $m = \lfloor \log_2(e) \rfloor + 1$. Let $H(e)$ represent the Hamming weight of e, i.e., $H(e) = \sum_{i=0}^{m-1} e_i$ is the number of ones in the binary expansion of e.

Definition: An *addition chain* U for a positive integer e of length l is a sequence of positive integers $U = \{u_0, u_1, \cdots, u_l\}$, and an associated sequence of r pairs.

$V = \{v_1, v_2 \cdots, v_l\}$ with $v_i = (i_1, i_2)$, $0 \leq i_2 \leq i_1 < i$, such that:

- $u_0 = 1$ and $u_l = e$;
- for each u_i,$1 \leq i \leq l$,$u_i = u_{i_1} + u_{i_2}$.

The shortest length of any valid addition chain for a given positive integer e is denoted as $l(e)$. Table I lists the set of exponents which have an optimal addition chain of length $l(e) = r$, for $r = 1, 2, \dots, 9$.

It is easy to get convinced that the search space for computing optimal addition chains increments its size rapidly. In fact, there exist $r!$ different and valid addition chains with length r. Obviously, the problem of finding the shortest ones becomes more and more complicated as r grows larger. Fig. 6 shows the first eight levels of the optimal addition chain tree.

Each of the deterministic heuristics outlined in Section II for the generation of addition chains clearly sets an upper bound on the function $l(e)$. In particular, the theoretical cost of the binary algorithm given in (3) implies that $l(e) \leq m + H(e) - 1$. A lower bound for $l(e)$ was found in [1] as, $\log_2 e + \log_2 H(e) - 2.13$. Therefore, we can write

$$\log_2 e + \log_2 H(e) - 2.13 \leq l(e) \leq \lfloor log_2(e) \rfloor + H(e) - 1. \tag{9}$$

TABLE I
Set of Exponents Which Have an Optimal Addition Chain of Length r

length r	Solutions
1	{2}
2	{3,4}
3	{5,6,8}
4	{7,9,10, 12,16,}
5	{11, 13, 14, 15, 17, 18, 20, 24, 32}
6	{19, 21, 22, 23, 25, 26, 27,28,30,33,34,36,40,48,64}
7	{29, 31, 35, 37, 38, 39, 41, 42, 43, 44,45,46,49,50,51,52, 54,56,60,65,66,68,72,80,96,128}
8	{47,53,55,57,58,59,61,62,63,67, 69,70,73,74,75,76,77,78,81,82, 83,84,85,86,88,90,92,97,98, 99,100,102,104,108,112,120,129, 130,132,136,144,160,192,256}
9	{71,79,87,89,91,93,94,95, 101,103,105,106,107,109,110,111,113,114, 115,116,117,118,119,121,122,123,124,125, 126,131,133,134,135,137, 138,140,145,146,147,148,149,150,152,153,154,156,161,162,163,164, 165,166,168,170,172,176,180,184,193,194,195,196,198,200,204,208, 216,224,240,257,258,260,264,272,288,320,384,512}

Let us suppose that we are interested in finding addition chains for several exponents of a given fixed bit-length, say, m. Then, as it was shown in [30], $l(e)$ is a function of the Hamming weight $H(e)$. Indeed, one can expect that on average $l(e)$ will be smaller for both, $H(e)$ closer to 0 and for $H(e)$ closer to m. On the contrary, when $H(e)$ is close to $m/2$, i.e., for those m-bit exponents having a balanced number of zeros and ones, $l(e)$ happens to be maximal [30].

Definition: An *addition sequence* is a generalization of an addition chain where not just one but several positive integers $e_1 < e_2 \dots < e_s$ must be included in the given sequence. It has been shown that the minimal length $l(e_0, e_1, \dots, e_s)$ of an addition sequence for $e_1, \dots, e_s$ is upper bounded by [20]

$$l(e_1, e_2, \dots, e_s) \leq \log e_s + (s + K)\frac{\log e_s}{\log \log e_s} \tag{10}$$

where K is a constant. For example, an addition sequence for {23, 28, 40, 47} is

$$1, 2, 3, 5, 10, 12, 13, \underline{23}, \underline{28}, \underline{40}, 45, \underline{47}. \tag{11}$$

Little is known about addition sequences bounds. However, it has been shown that finding a minimal length addition sequence is an **NP**-hard problem [20]. Some heuristics for generating optimal addition sequences are discussed next.

1) Generating Short Addition Sequences: Few heuristic methods able to generate reasonably short addition sequences have been reported [6], [34], [39].

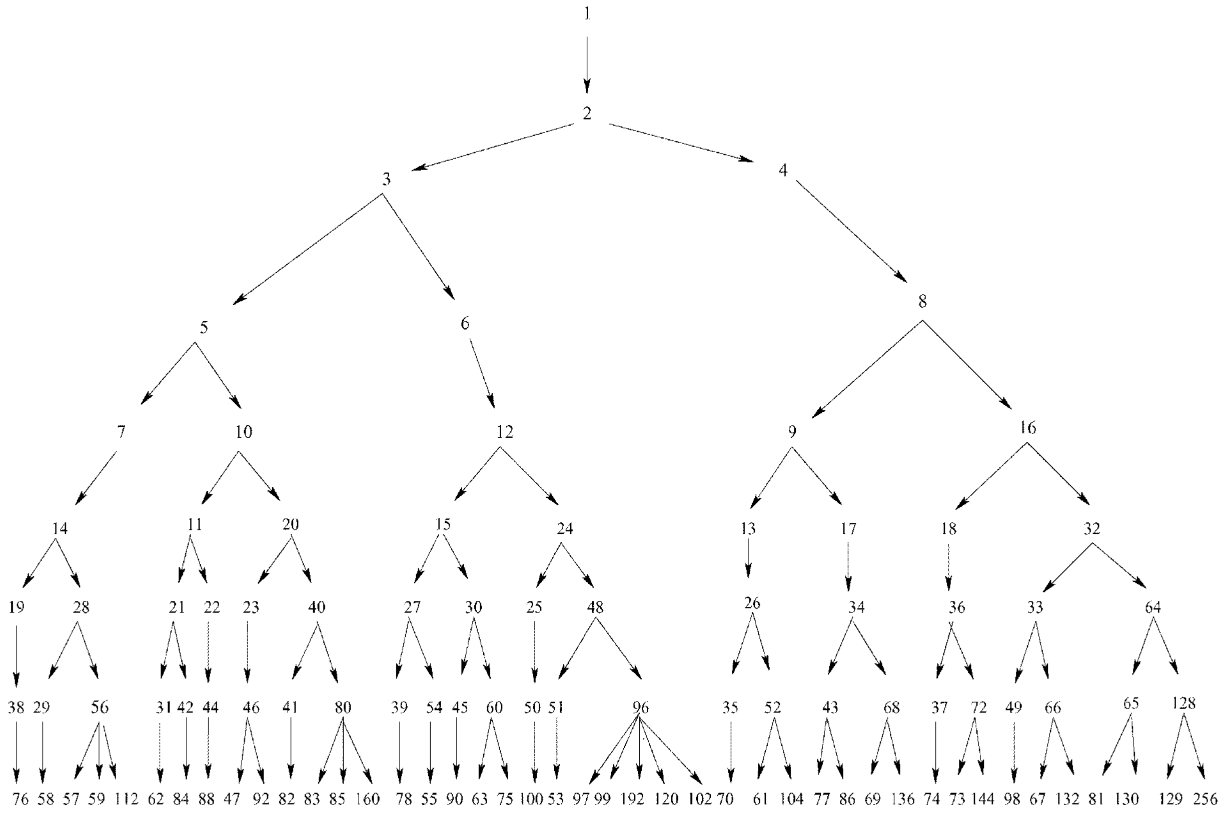

Fig. 6. Eight-level optimal addition chain tree.

The Bos–Coster method presented in [6], starts by defining a *protosequence* consisting of 1,2 together with the requested integers, i.e., $\{u_0 = 1, u_1 = 2, u_2 = e_1, u_3 = e_2, \cdots, u_{s+2} = e_s\}$. It then transforms this to the required sequence by using a heuristic composed by the following four methods.

1) *Approximation*. Let us suppose there are two elements already in the sequence such that $u_i + u_j = e_s - \epsilon$, with $u_i < u_j$ and ϵ positive and small. Then, insert $u_i + \epsilon$.
2) *Division*. If e_s is divisible by a small prime p, then aggregate $\{e_s/p, 2e_s/p, \ldots, (p-1)e_s/p, e_s\}$, in the sequence.
3) *Halving*. Let us suppose there is a small number $u_i = t$ already in the sequence such that, $e_s - t = 2^v K$, where v, K are both integers. Then, aggregate $\{e_s - t, (e_s - t)/2, \ldots, (e_s - t)/2^v\}$ in the sequence.
4) *Lucas*. Aggregate a *Lucas sequence* such that its last element is e_s.

The Bos–Coster method reports good experimental results when applied to 512-bit exponents, with Hamming weight of about two-thirds of 512. Nevertheless, the exact method of deciding which method should be used was left open (in fact, the authors in [6] mentioned that they unsuccessfully tried the simulated annealing technique as an optional method).

In this work, we implemented the insertion method (similar to the Bos–Coster Approximation method) shown in the algorithm of Fig. 7.

Let us suppose that we want to produce an addition sequence for an ordered set of s positive integers (windows), $\{e_1, e_2, \ldots, e_{s-1}, e_s\}$. First, the sets U, V, and W are initialized as shown in steps 2–3 of algorithm in Fig. 7. Notice that

```
Input:  An ordered set of $s$ integers
        $U := \{e_1, e_2, \ldots, e_{s-1}, e_s\}$ such that if
        $i < j$ then $e_i < e_j$
Output: An addition sequence for
        $\{e_1, e_2, \ldots, e_{s-1}, e_s\}$ with length $L$

Add_Seq_Generator($U, s$)
  1. $k = s - 2$;
  2. Set $U := \{u_1 = e_1, u_2 = e_2, \ldots, u_k = e_{s-2}\}$;
  3. $V := \{v_1 = e_{s-1}, v_2 = e_s\}$; $W = \emptyset$;
  4. while ($U \neq \emptyset$) do {
  5.     $\Delta = (v_2 - v_1)$;
  6.     $W := W \cup \{v_2\}$;
  7.     $\{v_1, v_2\}$ := max_two_elements($u_k, \Delta, v_1$};
  8.     if (($Delta < u_k$) && ($\Delta \notin U$)) then {
  9.         $U$ := Sort_Set($U \cup \{\Delta\}$);
  10.    } else if ($\Delta \in U$) then {
  11.        k = k-1;
  12.    }
  13. }
  14. L= Length_Set(W);
  15. output {W, L}
```

Fig. 7. An algorithm for generating short addition sequences.

TABLE II
AN EXAMPLE OF ADDITION SEQUENCE GENERATION

iteration	k	U	Δ	V	W
-	8	U:=$\{3,5,7,11,15,23,25,43\}$	-	V:=$\{93,147\}$	W:= $\emptyset$
1	8	U:=$\{3,5,7,11,15,23,25,43\}$	54	V:=$\{54,93\}$	W:= $\{147\}$
2	8	U:=$\{3,5,7,11,15,23,25,39\}$	39	V:=$\{43,54\}$	W:= $\{93,147\}$
3	7	U:=$\{3,5,7,11,15,23,25\}$	11	V:=$\{39,43\}$	W:= $\{54,93,147\}$
4	7	U:=$\{3,4,5,7,11,15,23\}$	4	V:=$\{25,39\}$	W:= $\{43,54,93,147\}$
5	7	U:=$\{3,4,5,7,11,14,15\}$	14	V:=$\{23,25\}$	W:= $\{39,43,54,93,147\}$
6	7	U:=$\{2,3,4,5,7,11,14\}$	2	V:=$\{15,23\}$	W:= $\{25,39,43,54,93,147\}$
7	7	U:=$\{2,3,4,5,7,8,11\}$	8	V:=$\{14,15\}$	W:= $\{23,25,39,43,54,93,147\}$
8	7	U:=$\{1,2,3,4,5,7,8\}$	1	V:=$\{11,14\}$	W:= $\{15,23,25,39,43,54,93,147\}$
9	6	U:=$\{1,2,3,4,5,7\}$	3	V:=$\{8,11\}$	W:= $\{14,15,23,25,39,43,54,93,147\}$
10	5	U:=$\{1,2,3,4,5\}$	3	V:=$\{7,8\}$	W:= $\{11,14,15,23,25,39,43,54,93,147\}$
⋮	⋮	⋮	⋮	⋮	⋮
16	W:= $\{1,2,3,4,5,7,8,11,14,15,23,25,39,43,54,93,147\}$				

the set $V := \{v_1 = e_{s-1}, v_2 = e_s\}$ is initialized with the two largest integers of the input set U. Thereafter, the main loop starts in step 4.

At each iteration, we compute the value $\Delta = v_2 - v_1$ in step 5. Then, we insert Δ into the sequence, thus guaranteeing that the addition of two sequence elements can produce v_2 (namely, $v_1 + \Delta$). As a consequence, the integer v_2 is added to the output set W (step 6). The set V is then updated with the two largest values among the three candidates: u_k, Δ and v_1 (step 7). Finally, in steps 8–12, if Δ is not already in U and if $\Delta < u_k$, then that element is added to the set U without distorting its ascending order (procedure *Sort_Set* in step 9). In the case that $\Delta \in U$, then the number of elements in U kept in the variable k, is decreased by one.

These iterations are repeated until the input set U is empty and, consequently, the output set W contains the required addition sequence.

An Example: Let us suppose that we want to produce an addition sequence for the following set of ten integers, {3, 5, 7, 11, 15, 23, 25, 43, 93, 147}. Table II describes how the sets U, V, W are being updated as the algorithm in Fig. 7 executes. The final addition sequence produced by our algorithm is then

$$W := \{1,2,\underline{3},4,\underline{5},\underline{7},8,\underline{11},14,\underline{15},\underline{23},\underline{25},39,\underline{43},54,\underline{93},\underline{147}\}. \tag{12}$$

The sequence in (12) is a valid addition sequence for the input set given. Notice that the sequence has a length of 16 elements.

According to our experiments, we found that the length of the sequences produced by the algorithm in Fig. 7 could be empirically upper bounded as

$$l(e_0, e_1, \ldots, e_s) \leq \frac{4}{3}\lfloor log_2(e_s)\rfloor + s + 1 \tag{13}$$

which is a slightly better value than the bound given in [6].

B. Sliding Window Method Using Addition Sequences

The pseudocode for the sliding window method using addition sequences is shown in Fig. 8. We use the same partitioning

```
Input:  x, n, e = $(e_{m-1} \ldots e_1 e_0)_2$

Output: y = $\mathbf{x}^e \bmod n$.

  1. Decompose $e$ into $\Psi$ zero and nonzero windows $W_i$ of length
     $L(W_i)$, for $i = 0,1,2,\ldots,\Psi-1$.
  2. Compute and store the addition sequence corresponding to the
     NZ nonzero windows found in the previous step, namely,
     $[W_0, W_1, \ldots, W_{NZ-1}]$
  3. y = $x^{W_{\Psi-1}}$;
  4. for $i = \Psi - 2$ downto 0 do {
  5.     y = $\mathbf{y}^{2^{L(W_i)}}$ ;
  6.     if $W_i \neq 0$ then y = $\mathbf{y} \cdot \mathbf{x}^{W_i}$;
     }
  7. output y
```

Fig. 8. Sliding window exponentiation using addition sequences.

algorithm described in the Section II-C, but taking advantage of the addition sequence concept, we may now allow much larger window sizes. Then, referring to Fig. 8, the following steps are performed.

- At step 1, the exponent e is partitioned using the strategy described in Section II-C (see Fig. 4). As a consequence, a total of Z zero windows and NZ nonzero windows will be produced.
- After having performed the partitioning phase, the next task of the algorithm consists of the computation of the addition sequence needed to obtain all the NZ nonzero windows found in the previous phase. This task can be accomplished at a cost of $l(W_0, W_1, \ldots, W_{NZ-1})$ preprocessing multiplications.

- At step 3, $\mathbf{y}$ is initialized using the value of the MSW as $\mathbf{y} = x^{W_{\Psi-1}}$. Notice that it is always assumed that $W_{\Psi-1} \neq 0$.
- At each iteration of the main loop, the power $\mathbf{y}^{2^{L(W_i)}}$ can be computed by performing $L(W_i)$ consecutive squarings. The total number of squarings is given as $m - L(W_{\Psi-1})$.
- At each iteration, one multiplication is performed whenever the ith word W_i is different than zero. Recall that NZ represents the number of nonzero windows. Therefore, the number of multiplications required at this step of this algorithm is precisely $NZ - 1$. Although the exact value of NZ will depend on the partitioning strategy instrumented, our experiments show that an approximate value for NZ using $q = 2$, $k = 5$, is about 0.15 m.

Thus, we find that the average number of multiplications needed to compute field exponentiation for an m-bit exponent e is given as

$$\begin{aligned} P(m,k,q) &= l(W_0,\ldots,W_{\Psi-1}) + m - L(W_{\Psi-1}) + NZ - 1 \\ &\approx l(W_0,\ldots,W_{\Psi-1}) + 1.15m - L(W_{\Psi-1}). \quad (14) \end{aligned}$$

From (14), it can be seen that one can optimize its computational cost by carefully selecting the most-significant-window $W_{\Psi-1}$. This feature will be exploited by the AIS heuristic to be explained in the next section.

Also, notice that the sliding window method requires, in general, the precomputation of the first 2^k odd powers of $\mathbf{x}$ (step 1 of Fig. 5) at a cost of $2^{k-1} - 1$ operations. In contrast, in the case of (14) that step is substituted by the computation of an addition sequence at a cost of $l(W_0,\ldots,W_{\Psi-1})$ operations, whose upper bound is given by (10) and (13).

Moreover, as we will see in the rest of this paper, the usage of a probabilistic heuristic on the computation of short addition sequences allows us to use much larger values of the parameter k implying a potential speedup on the computation of the field exponentiation operation.

IV. Artificial Immune System (AIS) and Problem Representation

In this section, we briefly discuss the main aspects that characterize AISs in general. Furthermore, we explain how the problem of finding short addition sequences for large exponents can be represented using an AIS setting.

A. Artificial Immune System (AIS)

The AIS is a relatively new computational intelligence paradigm which borrows ideas from the natural immune system (especially from the one corresponding to mammals) to solve relatively complex problems. In recent years, AIS has been successfully applied for solving problems in different areas such as computer and network security [2], [17], [22], fault detection [13], [19], scheduling [23], machine learning [16], [31] and optimization. Reported optimization problems solved by using AIS systems include multimodal [15], numerical [21], and combinatorial optimization [10].

From a biological point of view, the human immune system is a very complex system formed by a large number of cells and molecules and diverse mechanisms.

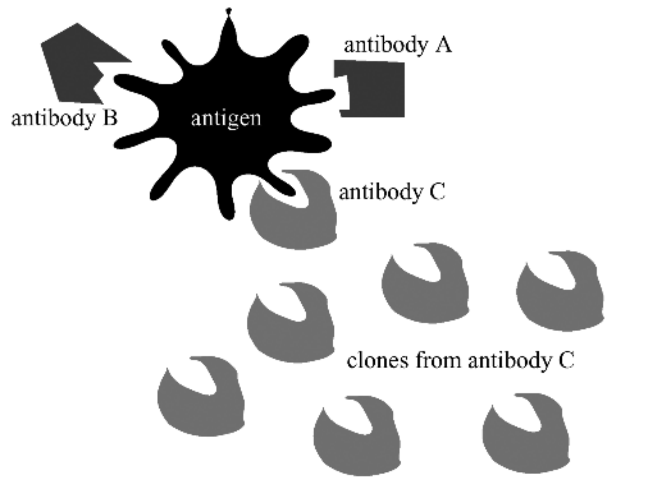

Fig. 9. The clonal selection principle of the immune system. Antibody C (the best affinity) is reproduced by cloning. The new clones will suffer a mutation process.

Some immunologists argue that one of the main functions of this system is to protect our bodies from the invasion of external microorganisms. It is composed of two defense lines: *innate* and *adaptive immunity*. Innate immunity is nonspecific which means that it is independent of the foreign antigen. The adaptive immunity has memory and learning capabilities and it is antigen-dependent, meaning that each different type of antigen will provoke a different immune response. The main components of the adaptive immunity are the cells called *B lymphocytes* or simply *B cells*. When B lymphocytes are stimulated by a specific antigen, they will produce a large number of molecules called *antibodies*, which play a major role in the adaptive immune response.

From the information processing perspective, the immune system is seen as a parallel and distributed adaptive system [18]. It is capable of learning; it uses memory and it is able of performing information associative retrieval. Particularly, it learns how to recognize patterns; it remembers patterns that have shown up in the past and its global behavior is an emergent property of many local interactions [12].

As previously mentioned, the immune system is a very complex system (probably its complexity is only comparable to that of the brain). However, for the sake of simplicity, we will only use two elements of the immune system in our model, namely, *antigens* (foreign microorganisms) and the *antibodies* (the main actors of the adaptive immune response).

The algorithm presented in this paper is based on a mechanism called *clonal selection principle* [8] that explains the way in which the antibodies eliminate a foreign antigen.[2] Such principle is explained in the next section.

B. Clonal Selection Principle

Fig. 9 depicts the clonal selection principle, which establishes the idea that only those antibodies that best match the antigen are stimulated. These stimulated antibodies are reproduced by

[2]Partially due to the fact that the immunology community has not yet entirely understood how the immune system works, the validity of the clonal selection principle is currently under debate (see, for example, [3] and [35]). However, in this work, it is shown that designing a heuristic inspired on that immunology principle appears to be the right choice for the optimization problem at hand.

cloning and the new clones suffer a mutation process with high rates (called hypermutation). After this process takes place, some of the newly created antibodies may increase their affinity to the foreign antigens. Those clones will increase the chances of neutralizing and/or eliminating the antigens. Once the foreign antigens have been exterminated, the immune system must return to its normal values, eliminating the exceeding antibodies cells (auto regulation).

However, some of the best cells remain in the body as memory cells. Then, in future encounters with the same kind of antigen (or a similar one), the immune system response will likely be more effective and efficient. This phenomenon is called secondary response.

Those antibodies showing lower affinity sometimes undergo receptor editing: Their low affinity receptors are replaced by new ones created randomly.

The processes of stimulation and cloning of the fittest antibodies, hypermutation and auto regulation are called the *clonal selection principle*. This is an oversimplification of what really happens in the natural immunity response. However, for the goals followed by most of the immunity-based artificial systems, such a simplification seems to be appropriate [15].

Hence, the immune aspects to be taken into account for modeling our algorithm are the following.

1) Stimulation of the higher affinity antibodies with respect to the antigen.
2) Cloning of the stimulated antibodies.
3) Proliferation rate proportional to antibodies' affinity.
4) Hypermutation rate inversely proportional to antibodies' affinity.
5) Receptor editing.
6) Immune memory.

Even this subset of immune mechanisms is still considerably complex as a large number of cells participate on them. Therefore, we will emulate these immune mechanisms using a simplified model of them as described in the remainder of this section.

C. Problem Representation

According to de Castro and Timmis [14], in every AIS, as in any other computational system with biological inspiration, the following elements must be defined.

- A representation of the system components.
- Evaluation mechanisms of individuals' interaction with their environment and/or with each other. The environment is usually stimulated by a set of input stimuli, one or more fitness functions, or by other means.
- Adaptation procedures that govern the dynamics of the system, i.e., how the system's behavior varies over time.

According to this framework, the elements of our algorithm were defined using the following setting.

- *A representation of the system components*. For the modular exponentiation problem, we defined two main actors: an antigen and an antibody population. A foreign antigen is represented as the exponent e that we wish to reach. Antibodies, on the other hand, are represented by the pair (U, l), where U is the addition chain sequence that contains the arithmetic recipe required for computing the desired goal (the antigen); and l is a positive integer representing the length of U, i.e., the number of steps needed to achieve the desired goal. The antibody population represents potential solutions for the problem in hand.
  For instance, if we wish to reach the antigen $e = 1903$, we may select the antibody $Ab = (U, l)$ composed by the addition chain sequence U

$$x^1 \to x^2 \to x^3 \to x^4 \to x^7 \to x^{14} \to x^{28} \to x^{29} \to x^{58} \to x^{116} \to x^{118} \to x^{236} \to x^{472} \to x^{475} \to x^{950} \to x^{1900} \to x^{1903}$$

  with length $l = 16$. Ab represents a feasible problem solution, i.e., an antibody with affinity value 16 (although this solution is not the best possible one, as it will be shown in the next section).
- *Evaluation mechanisms of individuals' interaction with their environment and/or with each other*. The affinity of a given antibody with respect to the antigen e is, therefore, equal to the length of its associated addition chain. The shorter the antibody's length is the better its associated affinity.
- *Procedures of adaptation that govern the dynamics of the system*: The dynamic of our system is based on the clonal selection principle.

Table III shows an analogy between some biological immune system elements on one side, and the way that those elements were modeled by our algorithm on the other.

TABLE III
ANALOGY BETWEEN THE BIOLOGICAL AND THE AIS DEFINED IN OUR ALGORITHM

Biological Immune System	Artificial Immune System
antigen	exponent e (from $B = A^e \bmod P$)
antibody	pair (U, l), where U is an addition chain of length l representing a potential solution that must be reached
antibody's affinity	length of the addition chain represented by the positive integer l (the shorter the better)
cloning	antibody's identical copies
hypermutation	changes applied on the clones
receptor editing	replacement of low affinity antibodies by new ones
immune memory and secondary response	accumulated knowledge consisting on solutions previously found and stored for different values of e

V. AIS HEURISTIC FOR FIELD EXPONENTIATION

In this section, we describe the AIS-based heuristic utilized in this paper for computing the field exponentiation operation. We first discuss the proposed AIS strategy algorithm. Then, two design examples that illustrate the algorithm behavior are explained in detail.

A. The AIS Heuristic

Next, we describe the AIS heuristic adopted in this work, considering the following aspects: antibody's construction, the hypermutation operator, the immune memory mechanism, and

```
Input:  A mutilated addition chain ($U = u_0, u_1, \cdots, u_{i-1}$), where $i$
        is the next position to be assigned and; the antigen $e$ that we
        want to reach.
Output: A complete addition chain ($U$) for $e$ with length $l$.

Fill($U, i, e$){
1. $j = i$;
2. while ($u_j \neq e$) do {
3.     if (Flip(F)) then
           use a doubling step if possible, i.e., $u_j = 2u_{j-1}$,
           provided that $u_j \leq e$. If $u_j > e$ go to 4.
4.     else if (Flip(0.5)) then
               set $u_j = u_{j-1} + u_{j-2}$
               provided that $u_j \leq e$. If $u_j > e$ go to 5.
5.         else do {
               set $u_j = u_{j-1} + u_k$, where $k$ is a randomly
               selected integer such that $0 \leq k < j$.
           } while ($u_j > e$)
6.     j =j+1
7. }
8. output ($U, j$) }
```

Fig. 10. Procedure for repairing a mutilated addition chain.

```
Input:  The antigen $e$ that we want to reach.
Output: A complete addition chain ($U = u_0, u_1, \cdots, u_l = e$) for $e$
        with length $l$.

Fresh_Ab($e$){
1. Set $u_0 = 1$ and $u_1 = 2$; (which implies $1 \to 2$)
2. Select 3 or 4 randomly and assign it to $u_2$
3. Complete the addition sequence by calling the
   procedure $(U, l) = FILL(U, 3, e)$
4. output $(U, l)$ }
```

Fig. 11. Algorithm that produces a complete addition chain.

the clonal selection algorithm. Finally, we present a complexity analysis of our algorithm.

1) Antibody's Construction: As explained in the previous section, in our system, an antibody is modeled by the pair (U, l), where U is a valid addition chain of length l for the antigen e. Therefore, we need to define a procedure able to build legal addition chains so that the system's antibody population can be created and mutated.

In order to see how this can be done, consider first the problem of completing a valid addition chain assuming that an in-progress (mutilated) addition chain $U = u_0, u_1, \cdots, u_{j-1}$, with $u_{j-1} < e$ has been already built. Under this scenario, one possibility for adding a new element in the chain would be to use the so-called *doubling step* [26], which is merely $u_j = 2u_{j-1}$. Notice that u_j would get the maximum possible value $2u_{j-1}$ that can be obtained from the in-progress addition chain, $U = u_0, u_1, \cdots, u_{j-1}$. However, it might be that $2u_{j-1} > e$, making illegal the usage of a doubling step. In that case, one can try instead $u_j = u_{j-1} + u_{j-2} < 2u_{j-1}$, which after the doubling step is the second maximum value that u_j can achieve from the given chain. However, even in this case, it is still possible that $u_j = u_{j-1} + u_{j-2} > e$. If that happens, one can try $u_j = u_{j-1} + u_k$, with k a randomly chosen integer such that $0 \leq k < j$.

Based on the above considerations,[3] we designed the algorithm shown in Fig. 10 as our main mechanism for producing legal addition chains for a given antigen e. Indeed, given the antigen e and an in-progress (mutilated) addition chain $U = u_0, u_1, \cdots, u_{j-1}$, with $u_{j-1} < e$, the procedure shown in Fig. 10 produces a complete addition chain able to achieve e in a fixed number of steps. Notice that our procedure utilizes a uniformly distributed random function $Flip(F)$. $Flip(F)$ accepts a parameter $F (0 \leq F \leq 1)$, and returns *true* with probability F or *false* in other case.

Using the algorithm of Fig. 10 as the main building block, the procedure of Fig. 11 produces a complete addition chain (antibody) for the antibody e.

2) The Hypermutation Operator: In nature, the hypermutation operator is inversely proportional to the clones' affinity, i.e., the higher the affinity of a clone, the lower its mutation rate and *vice versa*.

Notice that delicate perturbations in an addition chain can be introduced by placing a mutation point closer to the end of the addition chain (upper half). On the contrary, if the mutation point is placed closer to the beginning of the chain (lower half) the perturbation will be much more noticeable.

Based on this observation, the mutation operator was acting in a different section of the addition chain depending on the clone's affinity value. This way, clones showing high affinity were mutated in the upper half of the chain only. By contrast, those clones showing low affinity were mutated in the lower half of their chains. The algorithm of Fig. 12 shows the strategy followed for modeling the hypermutation operator.

3) Immune Memory: Previously found addition chains are stored in memory for future reference. Those solutions could be useful for future exponents. For instance, if the antigen is an even exponent e, then possibly the addition chain that had been found for $e/2$ could be useful by aggregating a single *doubling step* that doubles the last value of $e/2$, thus producing e (see steps 19–22 of the algorithm in Fig. 13).

4) The Clonal Selection Algorithm: The clonal selection algorithm for computing optimal addition chains is shown in Fig. 13. The parameters introduced in that algorithm are as follows.

- N is the number of antibodies to be created.
- P is the number of best antibodies which will be selected for cloning.

[3]That set of rules corresponds to a special class of addition chains known as *star chains* [26], [37].

```
Input:  A clone $Cl = (U, l)$ with affinity $l$, a region (either lower or
        upper half of the chain)
Output: A hypermutated clone $\hat{Cl} = (\hat{U}, \hat{l})$.

hypermutation($Cl, region$){
1. The mutation point $i$ for each clone is selected randomly
   within its region (either lower or upper half of the chain)
   corresponding to the clone's affinity.
2. Select a random number $j$ such that $0 \le j < i < e$.
3. The new (mutated) value of the clone's addition chain at
   the mutation point $u_{i+1}$ will be $u_{i+1} = u_i + u_j$, if it is
   possible, otherwise decrease $j$ until $u_{i+1}$ is a valid value.
4. Repair the upper part of the addition chain $\{u_{k>i+1}\}$
   by calling the function $(\hat{U}, \hat{l}) = FILL(U, i+2, e)$.
5. output $(\hat{U}, \hat{l})$
6. }
```

Fig. 12. The hypermutation operator.

- d is the quantity of low affinity antibodies that will be substituted.
- $iterations$ is the total number of iterations.

The above parameters must be defined by the user. However, based on a statistical study that we conducted, we suggest some values for them in the next section.

Referring to Fig. 13, the algorithm's dataflow can be summarized as follows. First in step 1, an initial population of N antibodies Ab_i for $i = 0, \cdots, N$ is created. The main loop of the algorithm starts immediately after (in step 3). In step 4, the N antibodies just created are sorted in ascending order according to their affinity values (i.e., their addition chain length). In step 5, only the best P antibodies are selected for cloning. The surviving P individuals are then ranked in ascending order according to their chain length (i.e., individuals with shorter chain lengths are ranked in the first place). The total number of clones to be created in steps 7–12 was determined according to the criterion suggested in [15]. This way, a total of N clones are generated from those antibodies ranked as the fittest ones. Next, individuals ranked in second place are allowed to produce a total of $N/2$ clones, those ranked in third place produce $N/3$ clones, etc. Therefore, the total number of clones T can be bounded as

$$\sum_{i=1}^{P} \text{round}\left(\frac{N}{i}\right) \le T \le P \cdot N \qquad (15)$$

where T is the total number of clones, N is the number of antibodies in the population, P are the selected antibodies (in general with different lengths) and round() rounds up its argument toward the closest integer. Each term of that sum corresponds to the number of clones to be generated for each selected antibody. If two or more antibodies share the same length, then the number of clones generated from them would be the same. In an extreme scenario, where all the antibody population has the same length, a total of $T = P \cdot N$ clones would be produced.

```
Input:  An exponent (antigen) $e$
Output: A quasi optimal addition chain (antibody) $U =$
        $u_0, u_1, \cdots, u_l = e$

AIS_Optimal_Addition_Chain($e$){
1.  for ($i = 1$ to $N$) do { /*Creating an initial population of $N$ antibodies */
        $Ab_i$ = Fresh_Ab($e$);
    }
2.  for ($i = 1$ to iterations) { /* Beginning of the main loop */
3.      Sort out the $N$ $Ab_i$ antibodies in ascending order according to their
    affinity values (i.e., chain lengths).
4.      Select the best $P$ antibodies (with different length values) from the
    antibodies population. Only those selected $P$ antibodies will be cloned.
5.      Define $C_i \in [1, P]$, for $i = 0, \cdots, P$ as the ranking index of each one of
    the $P$ antibodies.
6.  $k = 0$;
7.      for ($i = 1$ to $N$) do { /* Cloning */
8.          if ($C_i$ above average) then $region$ = upper half
9.          else $region$ = lower half
10.         for ($j = 1$ to $round\left(\frac{N}{C_i}\right)$) do {
11.             $Cl_k$ = hypermutation($Ab_i, region$);
12.             $k = k + 1$;
            }
        }
13.     Sort out the antibodies and newly created clones in ascending order.
14.     From the ordered set of $N$ original antibodies and $k$ hypermutated clones,
        select the $N$ top best and discard the rest.
15.     for ($i = N - d + 1$ to $N$) do { /* replacing the $d$ worst antibodies */
16.         $Ab_i$ = Fresh_Ab($e$);
        }
    } /* End main loop */
18. Select the antibody $B$ showing best affinity (shortest chain length).
19. if ($e$ is even && $e/2$ has already been computed) then {
20.     Set $M$ as the solution found for $e/2$.
21.     if ((length of ($M$)+1) < (length of $B$)) then {
22.         set $B = M$ adding one doubling step at the end of $B$.
        }
    }
23. Store $B$ in memory.
24. output $B$
25. }
```

Fig. 13. The clonal selection algorithm.

Notice that in step 11 a hypermutation operator is applied to each clone. As explained, this operator was designed (see algorithm in Fig. 12) so that the perturbation strength is inversely proportional to the individual's affinity.

After that, in step 13, the antibodies and clones just produced (N + T) are sorted in ascending order. From the ordered set of original antibodies and modified clones, only the top N individuals are selected, while the rest are discarded. Moreover, the d worst antibodies are replaced by brand new ones created through algorithm in Fig. 11. After updating individuals' ranking indexes, this process is repeated a predetermined number of iterations. At the end of the main loop, the best individual obtained is compared against previously computed and stored data (only in the case that e is an even integer).

5) Discussion: We summarize the rationale behind the algorithm of Fig. 13 as follows.

- We start by creating an initial antibody population whose members are seen as potential solutions. Because of the stochastic manner in which that antibody population is created (see algorithms in Figs. 10 and 11), it is expected that the antibody population will show a rich diversity of addition chains.
- We adopt higher cloning rates for those antibodies showing higher affinity.
- We carefully mutate the individuals assuring that those mutations will produce valid addition chains. Also, the hypermutation operator was designed in such a way that individuals showing high affinity would get relatively small perturbations, whereas individuals with low affinity are mutated much more aggressively.
- We favor higher affinity individuals by assuring the transmission of their information to the next generation (*elitism*).
- We periodically introduce brand new antibodies in order to maintain diversity in the population (thus emulating the *receptor editing* process).
- We use the algorithm's accumulated knowledge by consulting solutions previously found by the algorithm which were stored in a *memory*. That memory emulates the immune memory mechanism.

Although our clonal selection algorithm is clearly an oversimplified version of the real immune system, the aforementioned immune mechanisms adopted attempt to mimic what according to the clonal selection theory, is happening (at least partially) in biological immune systems. Moreover, our experimental results (to be discussed in the next section) suggest that the hypermutation operator together with the elitist mechanism do have a positive impact in the overall algorithm's performance, thus supporting the notion that each individual in our algorithm can be seen as a sort of "partial" recognizer able to transmit/share valuable information to the next generation of individuals.

Stepney *et al.* indicate in [35] that several approaches have been taken in the context of AIS, including the so-called *reasoning by metaphor*. The clonal AIS model adopted in this work, first proposed in [15], fits in that kind of AIS.

Moreover, notice that even though clonal AIS can be considered as very similar to the evolutionary algorithm (EA) model, both paradigms have some significant differences. Perhaps the most evident is the fact that in AIS there is no notion of the crossover operator so typically found in EAs. Conversely, in EAs, there is no cloning mechanism.

It is worth mentioning that often there exist quite a few optimal valid addition chains able to achieve the antigen e, with minimum length l. Thus, at the end of a given experiment, our clonal selection algorithm will typically produce several individuals tied in their affinity value.[4] This characteristic seems to be in synchrony with typical clonal AIS outputs, where the final result is an entire population of detectors [35].

6) Computational Cost of the AIS Strategy: Referring to the clonal selection algorithm of Fig. 13, we assess its computational cost as follows.

- The process of creating new antibodies (steps 1 and 17) carried out by algorithms in Figs. 10 and 11 is quite efficient. The cost of the algorithm in Fig. 11 is negligible. On the other hand, the computational cost of the algorithm $Fill(3)$ in Fig. 10 has a complexity per individual of $O(l)$, where l is the length of the produced addition chain. Based on (9), we can bound that length as

$$\log_2 e + \log_2 H(e) - 4.13 \leq l \leq \lfloor \log_2(e) \rfloor + H(e) - 3 \quad (16)$$

  where $H(e)$ is defined as the Hamming weight of the antigen e. A total of $N + d$ antibodies (steps 1 and 17) are generated per generation.
- Similarly, the hypermutation operator of step 11 is carried out by algorithms in Figs. 10 and 12. Notice that the hypermutation is quite similar to the process of creating new antibodies. The only difference is that the algorithm of Fig. 12 just needs to produce part of the addition chain. Therefore, the computational cost of this operator per individual is also $O(l)$. A total of T clones [see (15)] are hypermutated (step 11) per generation.
- The sorting process of N antibodies (step 2), $N + T$ antibodies and clones (step 13) and N surviving antibodies (step 18) per generation can be carried out at a computational cost of about $O((3N + T)log(3N + T))$.

Therefore, the total computation cost per iteration of the clonal selection algorithm in Fig. 13 is given as

$$\text{Cost} = O\left(l(N + d + T)\right) + O\left((3N + T)\log(3N + T)\right). \quad (17)$$

Incidentally, it is worth mentioning that the computational effort required for the computation of field exponentiation itself is considerably more expensive than the above estimation for the clonal selection algorithm. Field exponentiation has an estimated complexity of $O(n^3)$ bit operations [32].

B. A Design Example for a Small Exponent

The exponent e is named the antigen or goal that the AIS is trying to achieve. Starting with an initial population of N antibodies, the algorithm uses the cloning mechanism to generate slightly different replicas that are then selected based on the fitness of the individuals. As previously mentioned, clone fitness is measured in terms of the length of its corresponding addition chain. In order to illustrate how our algorithm computes its task, let us consider the case when we want to obtain an optimal addition chain for our running example, the antigen $e = 1903$.

[4]For the purposes of efficient field exponentiation computation, all addition chains having a minimum length are, in general, equally valuable.

Example: Given the antigen $e = 1903$, the algorithm of Fig. 13 performs as follows.

1) An initial population of N antibodies Ab is constructed using algorithms of Figs. 10 and 11. For instance, let us suppose that the third antibody generated $Ab_3(U, l)$ has an addition chains given as, 1-2-4-8-16-24-48-49-98-196-294-588-612-1224-1836-1844-1893-1901-1903; with an associated affinity equal to 18.
2) Sort out the antibody population in ascending order according to the affinity values.
3) Select the best P antibodies (with different length values) from the antibody population. Only the P selected antibodies will be cloned.
4) Using (15), determine the total number of clones (T) to be generated for selected antibodies. To give a concrete example, consider $N = 30$, $P = 7$. Let us say that after ranking the P best clones, we observe that Ab_3, and Ab_{29} are tied in the first place sharing the same shortest chain length; Ab_2, Ab_{23}, and Ab_{17} rank in the second, third, and fourth places, respectively, and that Ab_{11} and Ab_{13} are tied in the fifth (last) place. Therefore, the ranking indexes C_i, for $i = 0, \cdots, 7$ would be 1, 1, 2, 3, 4, 5, 5, respectively. Thus, the total of clones to be created would be

$$T = \sum_{i=1}^{7} \text{round}\left(\frac{30}{i}\right) = 30+30+15+10+7+6+6 = 104.$$

5) Create the clones for the selected antibodies.
6) Apply the hypermutation operator to each clone (see the algorithm in Fig. 12).
   (a) For instance, a clone generated from the highest affinity individual Ab_3 will get a mutation point selected from the upper half of its chain. Let us say that this point is $i = 14$ (step 1, algorithm from Fig. 12).
   (b) A random number j is selected, $0 \leq j < i < e$, for example, $j = 7$ (step 2, the algorithm from Fig. 12)
   (c) The new value of the clone's addition chain at the mutation point u_{i+1} will be $u_{i+1} = u_i + u_j$, then we have $U_{15} = 1836 + 49 = 1885$, to this point our chain is the following: 1-2-4-8-16-24-48-49-98-196-294-588-612-1224-1836-1885
   (d) Repair the upper part of the addition chain $\{u_{k>i+1}\}$ with $FILL(k)$. Suppose the resulting addition chain is: 1-2-4-8-16-24-48-49-98-196-294-588-612-1224-1836-1885-1901-1903 with affinity $l = 17$.
7) Compute the associated affinity values for the T mutated clones.
8) From the set of original antibodies and modified clones, select the N top best and discard the rest.
9) Replace the d antibodies showing less affinity by new ones. For example, let us say that one of the brand new individuals so produced is: 1-2-3-6-12-15-30-33-66--99-198-213-426-852-885-1737-1836-1869-1902-1903
10) Compute the associated affinity values for the d new individuals. Notice that the affinity value for our new antibody is 19.
11) Go to step 3, a predetermined number of $iterations$.
12) The best antibody $\mathbf{B}$ is selected.
13) As $e = 1903$ is not even, then go to the next step.
14) Store $\mathbf{B}$ in *memory*.
15) Report $\mathbf{B}$ as the best solution found.

As a result of executing the above algorithm, our AIS-based heuristic was able to find several addition chains of length $l = 15$ for the exponent $e = 1903$. For example

$$\begin{aligned} x^1 &\rightarrow x^2 \rightarrow x^3 \rightarrow x^6 \rightarrow x^{12} \rightarrow x^{24} \rightarrow x^{25} \rightarrow x^{50} \\ &\rightarrow x^{100} \rightarrow x^{200} \rightarrow x^{300} \rightarrow x^{600} \rightarrow x^{900} \rightarrow x^{1800} \\ &\rightarrow x^{1900} \rightarrow x^{1903} \qquad (18) \\ x^1 &\rightarrow x^2 \rightarrow x^3 \rightarrow x^5 \rightarrow x^8 \rightarrow x^{13} \rightarrow x^{26} \rightarrow x^{39} \\ &\rightarrow x^{78} \rightarrow x^{156} \rightarrow x^{312} \rightarrow x^{624} \rightarrow x^{936} \rightarrow x^{1872} \\ &\rightarrow x^{1898} \rightarrow x^{1903}. \qquad (19) \end{aligned}$$

Let us recall that in Sections II-A and II-B it was found that for $e = 1903$ the binary, quaternary, octal, and hexa methods find addition chains of length 18,17, 16, and 16, respectively. It is worth to remark that the shortest addition chain for $e = 1903$ is precisely $l(e) = 15$ [26].

C. AIS Heuristic for Large Exponents

It is not advisable to directly apply the AIS heuristic for the computation of addition chains when dealing with large exponents. This is due to the fact that as the exponent bit-length grows larger, the addition chain length attained by our AIS heuristic tends to significantly deviate from the optimal and/or best-known values.

Fortunately, we can use instead the sliding window method described in Section III-B. Under this scenario, the concept of exponent partitioning described in Section II-C together with the concept of *addition sequences* described in Section III-A1 will emerge as the most important tools for generating quasi-optimal addition chains for large exponents.

In that regard, consider the algorithm shown in Fig. 14. Let us recall that the strategy followed here for large exponents can be divided into two main phases: exponent partitioning and addition sequence generation.

Referring to Fig. 14, the procedure $AIS\_Add\_Seq\_Large\_Exp$, takes as inputs an m-bit exponent e to be processed and the parameter $MaxW\_MSW$, which establishes the maximum size that the MSW can take in the partition phase. By default, the minimum size for MSW is 6. At each iteration, the i most significant bits of e are assigned to the variable MSW (see step 3). In step 4, the $m - i$ least significant bits of the exponent e are assigned to the auxiliary variable e_aux. Then, in step 5, an optimal addition chain A for MSW is obtained through a call to the AIS algorithm of Fig. 13 previously discussed.

In step 6, e_aux is partitioned using the strategy described in Section II-C and depicted in Fig. 4. As a consequence, a total of Z zero windows and NZ nonzero windows will be produced. After having sorted in step 7 all the NZ nonzero windows, a suitable element a in the addition chain A, greater than W_{NZ-1} is added. Then, an addition sequence for the set $U = \{W_0, W_1, \ldots, W_{NZ-1}, a\}$ is produced, by invoking the procedure of Fig. 7.

```
Input: $e = (e_{m-1}e_{m-2}\dots e_1e_0)_2, Max\_MSW$

Output: Addition Sequence for $e$.

AIS_Add_Seq_Large_Exp($e, MaxW\_MSW$)
1.  $N\_Op = 0$; Minimum = $\infty$;
2.  for $i = 6$ to $MaxW\_MSB$ do {
3.      $MSW = (e_{m-1}e_{m-2}e_{m-3}\dots e_{m-i})_2$;
4.      $e\_aux = (e_{m-(i+1)}e_{m-(i+2)}\dots e_1e_0)_2$;
5.      $A$ = AIS_Optimal_Addition_Chain($MSW$);
6.      Decompose $e\_aux$ into a total of $\Psi$ windows $W_i$,
        for $i = 0,1,2,\dots,\Psi-1$, with $\Psi = Z + NZ$.
        A total of $Z$ zero windows and $NZ$ nonzero
        windows are produced.
7.      Sort all $NZ$ nonzero windows so produced
        in ascending order, $U=\{W_0, W_1, \dots, W_{NZ-1}\}$
8.      Select a suitable element in $a \in A$ such that
        $a > W_{NZ-1}$.
9.      $U=\{W_0, W_1, \dots, W_{NZ-1}, a\}$;
10.     $\{Seq, L\} = Add\_Seq\_Generator(U, NZ+1)$;
11.     Compute the number of operations $N\_Op$ needed;
12.     if ($N\_Op < Minimum$) then {
13.         Store the pair of sequences: (Seq, A);
14.         $Minimum = N\_Op$;
15.     }
16. }
output (Seq, A, Minimum)
```

Fig. 14. Finding short addition sequences for large exponents.

At this point, the algorithm is able to estimate the expected number of operations N_Op needed for computing the field exponentiation operation by applying (14). If the algorithm determines in step 12 that the sequences Seq, A have associated a minimum number of operations it proceeds to store them. Otherwise, it continues with the next iteration. After having examined all possible candidates for MSW in the range from 6 to $MaxW\_MSW$ bits, the algorithm in Fig. 14 outputs the pair of sequences $\{Seq, A\}$ that optimize field exponentiation. The dataflow of this algorithm is illustrated with a design example in the next section.

1) A Design Example: Let us consider the following design example for the 128-bit exponent given as, $e = (DCC99E15F158F280B81583CC8CC5D2CF)_{16}$, with $m = 128$ bits, and a Hamming weight $H(e) = 62$.

Partitioning: As discussed in Section III-B, the strategy followed for the exponent partitioning consisted on allowing a large MSW followed by relatively small windows, being the main idea to try to minimize the second component of (14). We consider all possible candidates for MSW in the range from 6 to $MaxW\_MSW = 20$ bits, and at the same time, we fixed the maximum size allowed for all the other nonzero windows to $k = 6$. We also fixed the maximum value of consecutive zeros to $q = 2$. Then, we invoked the algorithm in Fig. 14 in order to find the best MSW.

As a result, our algorithm came out with a partitioning consisting of a 17-bit MSW, namely $(1B993)_{16}$ followed by 15 nonzero windows distributed as shown below:

$$\underbrace{11011100110010011}_{1B993}00\underbrace{1111}_{F}0000\underbrace{101011}_{2B}\underbrace{111}_{7}000$$
$$\underbrace{101011}_{2B}000\underbrace{1111}_{F}00\underbrace{101}_{5}0000000\underbrace{10111}_{17}000000$$
$$\underbrace{101011}_{2B}00000\underbrace{1111}_{F}00\underbrace{11001}_{19}000\underbrace{11}_{3}00\underbrace{11}_{3}000$$
$$\underbrace{1011101}_{5D}00\underbrace{1011}_{B}00\underbrace{1111}_{F}.$$

Notice that the nonzero windows obtained from the partitioning phase are all odd and none of them (except for the very first window) contains two or more consecutive zeros.

Addition Sequence: We must derive a short addition sequence for all the nonzero window values found in the previous step. Note that we only need to consider ten different values as some windows appear several times in the partitioned exponent shown above. Hence, we need to find a short addition sequence for the following window values:

$$\{3, 5, 7, B, F, 17, 19, 2B, 5D, 1B993\}_{16} \equiv \{3, 5, 7, 11, 15, 23, 25, 43, 93, 113043\}.$$

As explained previously, the algorithm of Fig. 14 finds first a nearly optimal addition chain for MSW. The following 20-step addition chain for MSW $= (1B993)_{16} \equiv 113043$ was obtained:

$$1 \to 2 \to 3 \to 6 \to 9 \to 18 \to 36 \to 72 \to 144 \to 147 \to 294 \to 588 \to 1176 \to 2352 \to 4704 \to 9408 \to 18816 \to 37632 \to 75264 \to 112896 \to \underline{113043}. \tag{20}$$

Notice that in the above addition chain, the target value MSW $= 113043$ is obtained as $112896 + 147 = 113043$. Because of that, in step 7 of Fig. 14, the value $a = 147$ is chosen. Now, we need to find a short addition sequence for the ordered set, {3, 5, 7, 11, 15, 23, 25, 43, 93, 147}.

However, this was the example analyzed earlier in Section III-A1, where according to (12), the following 16-step solution was found after using the algorithm from Fig. 7:

$$W := \{1, 2, \underline{3}, 4, \underline{5}, \underline{7}, 8, \underline{11}, 14, \underline{15}, \underline{23}, \underline{25}, 39, \underline{43}, 54, \underline{93}, \underline{147}\}.$$

Further optimizations in the above solution combined with the rest of the addition sequence for MSW yielded the following 26-step addition sequence:

$$1 \to 2 \to \underline{3} \to 4 \to \underline{5} \to \underline{7} \to \underline{11} \to \underline{15} \to 18 \to \underline{23} \to \underline{25} \to \underline{43} \to 54 \to 86 \to \underline{93} \to \underline{147} \to 294 \to 588 \to 1176 \to 2352 \to 4704 \to 9408 \to 18816 \to 37632 \to 75264 \to 112896 \to \underline{113043}. \tag{21}$$

TABLE IV
NUMBER OF OPERATIONS USING SEVERAL METHODS FOR EXAMPLE 5.3.1

Strategy	**Number of Operations**
Binary	188
Quaternary	171
Octal	170
Hexa	164
Sliding window ($k = 5, q = 2$)	155
Sliding window ($k = 6, q = 2$)	154
AIS Sliding window ($k = 6, q = 2$)	152

Number of Operations: Referring to the algorithm shown in Fig. 8 and its complexity analysis summarized in (14), the number of arithmetic operations required for computing the field exponentiation is given as follows.

- A total of 26 multiplications needed to generate the addition sequence specified in step 2 of Fig. 8 and depicted by (21).
- A total of $m - L(W_{\Psi-1}) = 128 - 17 = 111$ squarings corresponding to step 5 of the algorithm in Fig. 8.
- A total of 15 multiplications in order to combine all the $\mathrm{NZ} - 1 = 15$ intermediate nonzero window values corresponding to step 6 in Fig. 8.

Therefore, we conclude that the total number of arithmetic operations for this example is given as

$$P(m, k, q) = 26 + 111 + 15 = 152. \qquad (22)$$

It is customary to use the ratio $P(m, k, q)/m$ as a figure of merit for field exponentiation [27], [30]. For our working example, the achieved ratio is of about

$$\frac{P(m, k, q)}{m} = 1.1875. \qquad (23)$$

We show in Table IV the number of operations obtained by the binary, quaternary, octal, and hexa methods discussed in Sections II-A and II-B. We also present the number of operations required by the sliding window method without AIS using $k = 5$, 6 and $q = 2$ as indicated in the Table. It can be seen that for this specific example, the AIS sliding window technique yields the lowest number of operations.

VI. EXPERIMENTS AND STATISTICAL TESTS

In this section, we present experimental results obtained from several relevant statistical tests performed to our algorithm. Then, we compare the AIS heuristic against some traditional deterministic strategies. At the same time, working with a family of exponents particularly hard to optimize, we also report complete solutions for their associated shortest addition chains.

A. Variance Analysis

In order to assess the algorithm's sensitivity to its parameters, we conducted an analysis of variance (ANOVA). The parameters analyzed were as follows.

- N: the number of antibodies to be created.
- P: the number of best antibodies which will be selected for cloning.
- d: the quantity of low affinity antibodies that will be substituted.
- F: a random variable ($0 \leq F \leq 1$) that selects which rule to apply during the process of antibody's construction (see the algorithm in Fig. 10).

The above parameters were considered the independent variables, while the dependent variable was the length of the addition chain found by the algorithm.

We chose three different values (levels) for each of the mentioned parameters. The tested levels were as follows.

- N: (15), (30), and (45).
- P: (N/1), (N/2), and (N/4).
- d: (0.0 of N), (0.1 of N), and (0.2 of N).
- F: (0.5), (0.7), and (0.9).

The experiment consisted on executing 30 independent runs of the algorithm with each different combination of the parameters levels. Therefore, we performed a total of 2430 runs of the algorithm. With the aim of performing balanced comparisons, we set the parameter $iterations$ such that the number of calls to the function $FILL()$ were the same for all the experiments. From that variance analysis, we can conclude that:

- The probability that the effect of the parameter N is due to the random processes is less than 0.01.
- The probability that the effect of the parameter P is due to the random processes is less than 0.01.
- The parameter d does not have any effect on the algorithm, its effect is product of the random processes.
- The probability that the effect of the parameter F is due to the random processes is less than 0.01.

Therefore, the parameters N, P, and F do have a real effect on the algorithm's performance.

1) Parameters Values Suggested: Based on the statistical study performed, we can suggest the following values for the parameters used in this algorithm:

- N: Number of antibodies: Use $N \in [30, 45]$.
- P: selected antibodies: Use $N/4$.
- d: replaced antibodies: 0.1% of the total population.
- F: Use $F = 0.7$.

B. Accumulated Addition Chain Lengths for Small Exponents

In [5], a method based upon continued fraction expansion for computing short addition chains was presented. Using their algorithm as a general framework, the authors tested the performance obtained by several traditional addition-chain generator strategies, such as the binary and quaternary methods, dichotomic, dyadic, total, Fermat, and the factor methods. A description of those methods can be found in [5] and [26]. Then, for each selected strategy, authors reported the total accumulated addition chain lengths for all exponents $e \in [1, 1000]$.

As a preliminary test for our heuristic, we repeated the same experiment reported in [5] but this time using our own strategy as a search engine.

All the results obtained with the AIS approach reported in this section were obtained applying the following parameter values.

- population size $N = 45$.
- selected antibodies $P = 0.25 * N$.
- replaced antibodies $d = 0.1 * N$.

TABLE V
ACCUMULATED ADDITION CHAIN LENGTHS FOR ALL EXPONENTS $e \in [1, 1000]$ (COMPARISON AMONG DIFFERENT HEURISTICS)

Optimal value=**10808**	
Strategy	**Total length**
Dyadic [5]	10837
Total [5]	10821
Fermat [5]	10927
Dichotomic [5]	11064
Factor [5]	11088
Binary	11925
Quaternary	11479
Artificial Immune System heuristic	
Best	10813
Average	10818.5
Median	10818.5
Worst	10825
Std. Dev.	3.06

- $F = 0.7$.
- *iterations*$= 25$.

The statistical results were obtained from 30 independent runs of the algorithm.

Table V compares the heuristics accumulative addition chain reported in [5] against the one obtained by our AIS heuristic. It can be seen that compared with all other featured strategies, our algorithm was able to compute the best approximation to the optimal value (which was obtained by enumeration), with a percentage error rather negligible (less than 0.07%).

Furthermore, we expanded this experiment using larger exponents. Tables VI and VII show accumulative addition chain lengths obtained by our heuristic for exponents less than 512, 1024, 2000, 2048, and 4096, respectively. For comparative purposes, we included the optimal value and the value corresponding to the binary and quaternary method.

Once again, although the AIS strategy could not find all the optimal values, its percentage error was less than 0.4% for all cases considered. That low error rate implies that for any given fixed exponent e with $e < 4096$, our strategy would be able to find the requested shortest addition chain in at least 99.6% of the cases.

Table VIII shows the AIS computational time for several exponent lengths. We used the gcc compiler running under i686-linux operating system in a UltraSPARC II at 450 MHz. It is noticed that our experimental results show a reasonable match with the computational costs predicted by (17).

Additionally, we collected the associated uncertainty of our results through the computation of the experiments' confidence intervals. This was done by applying a bootstrap resampling statistical test. The average ranges for each set of experiments are shown in Table IX with a confidence interval of 95% after executing 30 independent runs using different random seeds.

TABLE VI
ACCUMULATED ADDITION CHAIN LENGTHS FOR ALL EXPONENTS LESS THAN 512 ($e \in [1, 512]$) AND 1024 ($e \in [1, 1024]$)

e:	$[1, 512]$	$[1, 1024]$
Optimal:	4924	11115
Binary:	5388	12301
Quaternary	5226	11862
	AIS results	
Best	4924	11120
Average	4925.03	11126.433
Median	4925	11126.00
Worst	4927	11132
Std. Dev.	0.89	3.014

TABLE VII
ACCUMULATED ADDITION CHAIN LENGTHS FOR ALL EXPONENTS $e \in [1, 2000]$, $e \in [1, 2048]$, AND $e \in [1, 4096]$

e :	$[1, 2000]$	$[1, 2048]$	$[1, 4096]$
Optimal:	24063	24731	54425
Binary:	26834	27662	61455
Quaternary	25923	26664	58678
	AIS results		
Best	24108	24778	54617
Average	24120.20	24792.2	54644.033
Median	24120.0	24791.5	54640
Worst	24133	24807	54674
Std. Dev.	5.88	6.094	12.053

TABLE VIII
AIS COMPUTATIONAL TIME FOR SEVERAL EXPONENT BIT LENGTHS

e length in bits	timing (in milliseconds)
12	145.8
14	150.6
16	156.0
18	161.7
20	166.8

The importance of performing this type of test lies in the fact that only by using statistical tests can one reasonably ensure that the results yielded by a probabilistic heuristic are consistent and independent of the random seed used. This way, Table IX provides statistical evidence that the experimental lower and upper average values are very close to each other. Thus, it is fair to say that the average algorithm behavior is quite similar from one execution to the other, which is a desirable feature for a probabilistic heuristic to exhibit.

TABLE IX
AVERAGE WITH 95% CONFIDENCE FOR RESULTS OBTAINED BY THE AIS (30 INDEPENDENT RUNS)

	Average	
e	from	to
512	4924.7	4925.4
1000	10817	10820
1024	11125	11128
2000	24118	24122
2048	24790	24794
4096	54640	54649

C. *A Special Class of Exponents Hard to Optimize*

Let $e = c(r)$ be the smallest exponent that can be reached using an addition chain of length r. Solutions for that class of exponents are known up to $r = 30$ and a compilation of them can be found in [11]. Interestingly enough, the computation difficulty of finding the shortest addition chains for those exponents seems to be among the hardest if not the hardest ones of studied exponent families [26].

In order to assess the actual power of the AIS strategy as a search engine, we used it to generate all the shortest addition chains of the exponents $c(r)$ for $r = 0, 1, 2, \ldots, 30$.

In all cases considered, our AIS heuristic was able to generate a valid addition chain having the predicted optimal length. Notice that the search space size for this special class of exponents (considering both feasible and infeasible individuals) is $r!$. Hence, in the case of $r = 30$, finding the shortest addition chain for the exponent $c(r = 30) = 14143037$, implied to search over a space whose approximate size is

$$r! = 30! = 265252859812191058636308480000000 \approx 2^{107}.$$

VII. APPLICATIONS

Some practical applications of addition chains are described in this section. First, in Section VII-A, the efficient computation of multiplicative inverses based on optimal addition chains is explained. The material included in that section closely follows the discussion presented in [9]. Then, in Section VII-B, the combination of the AIS heuristic together with the sliding window method for computing large exponentiation is presented.

A. *Optimal Addition Chains for Computing Multiplicative Inverses*

Among the basic field arithmetic operations, namely, addition, subtraction, multiplication, and inversion of nonzero elements, the later is the most time-consuming one. The multiplicative inversion of an element $A \in F$ consists on finding an element $A^{-1} \in F$ such that $A \cdot A^{-1} \equiv 1 \bmod P(x)$. Several algorithms for computing multiplicative inverses over binary extension fields $F = \mathrm{GF}(2^n)$ have been proposed in the specialized literature [25], [36], [41].

One well-known strategy is based on Fermat's Little Theorem (FLT) which establishes that for any nonzero element $A \in$

TABLE X
SHORTEST ADDITION CHAINS FOR A SPECIAL CLASS OF EXPONENTS (TABLE 1 OF 3)

exponent $e = c(r)$	Addition Chain	Length r
1	1	0
2	$1 \to 2$	1
3	$1 \to 2 \to 3$	2
5	$1 \to 2 \to 4 \to 5$	3
7	$1 \to 2 \to 4 \to 6 \to 7$	4
11	$1 \to 2 \to 4 \to 8 \to 10 \to 11$	5
19	$1 \to 2 \to 4 \to 8 \to 16 \to 18 \to 19$	6
29	$1 \to 2 \to 4 \to 8 \to 16 \to 24 \to 28 \to 29$	7
47	$1 \to 2 \to 3 \to 6 \to 12 \to 15 \to 30 \to 45 \to 47$	8
71	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 64 \to 68 \to 70 \to 71$	9
127	$1 \to 2 \to 4 \to 8 \to 9 \to 18 \to 36 \to 54 \to 108 \to 126 \to 127$	10
191	$1 \to 2 \to 3 \to 6 \to 9 \to 18 \to 27 \to 54 \to 108 \to 162 \to 189 \to 191$	11
379	$1 \to 2 \to 4 \to 8 \to 16 \to 18 \to 36 \to 54 \to 108 \to 162 \to 324 \to 378 \to 379$	12
607	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 102 \to 204 \to 408 \to 510 \to 606 \to 607$	13
1087	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 120 \to 240 \to 360 \to 720 \to 1080 \to 1086 \to 1087$	14
1903	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 160 \to 180 \to 340 \to 520 \to 1040 \to 1560 \to 1900 \to 1903$	15
3583	$1 \to 2 \to 3 \to 6 \to 12 \to 18 \to 36 \to 72 \to 144 \to 288 \to 576 \to 594 \to 1188 \to 2376 \to 3564 \to 3582 \to 3583$	16
6271	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 192 \to 384 \to 768 \to 1536 \to 1537 \to 3074 \to 6148 \to 6244 \to 6268 \to 6271$	17
11231	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 30 \to 50 \to 100 \to 200 \to 400 \to 800 \to 1600 \to 3200 \to 6400 \to 9600 \to 11200 \to 11230 \to 11231$	18
18287	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 160 \to 320 \to 640 \to 1280 \to 1283 \to 2563 \to 3846 \to 7692 \to 15384 \to 17947 \to 18267 \to 18287$	19

$\mathrm{GF}(2^n)$, the identity $A^{-1} \equiv A^{2^n-2}$ holds. As surprising as it may sound, this means that multiplicative field inversion can be computed via an exponentiation operation.

Noticing that the exponent $e = 2^n - 2$ can equivalently be expressed as $e = \sum_{i=1}^{n-1} 2^i$, we can write

$$A^{-1} = A^{2^n-2} = A^{\sum_{i=1}^{n-1} 2^i} = \prod_{i=1}^{n-1} A^{2^i} = A^{2^1} \cdot A^{2^2} \cdots A^{2^{n-1}}. \quad (24)$$

TABLE XI
SHORTEST ADDITION CHAINS FOR A SPECIAL CLASS OF EXPONENTS
(TABLE 2 OF 3)

exponent $e = c(r)$	Addition Chain	Length r
34303	$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 14 \rightarrow 28 \rightarrow 56$ $\rightarrow 112 \rightarrow 224 \rightarrow 252 \rightarrow 504 \rightarrow 1008 \rightarrow 2016$ $\rightarrow 4032 \rightarrow 8064 \rightarrow 16128 \rightarrow 32256 \rightarrow 34272$ $\rightarrow 34300 \rightarrow 34303$	20
65131	$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 10 \rightarrow 11 \rightarrow 22 \rightarrow 44 \rightarrow 88$ $\rightarrow 132 \rightarrow 220 \rightarrow 440 \rightarrow 880 \rightarrow 1760$ $\rightarrow 3520 \rightarrow 7040 \rightarrow 14080 \rightarrow 28160$ $\rightarrow 56320 \rightarrow 63360 \rightarrow 65120 \rightarrow 65131$	21
110591	$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 10 \rightarrow 20 \rightarrow 40 \rightarrow 80$ $\rightarrow 160 \rightarrow 320 \rightarrow 640 \rightarrow 1280 \rightarrow 2560 \rightarrow 2570$ $\rightarrow 5140 \rightarrow 10280 \rightarrow 20560 \rightarrow 23130 \rightarrow 43690$ $\rightarrow 87380 \rightarrow 110510 \rightarrow 110590 \rightarrow 110591$	22
196591	$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 48$ $\rightarrow 96 \rightarrow 99 \rightarrow 195 \rightarrow 390 \rightarrow 780$ $\rightarrow 1170 \rightarrow 2340 \rightarrow 4680 \rightarrow 9360 \rightarrow 18720$ $\rightarrow 18726 \rightarrow 37446 \rightarrow 74892 \rightarrow 149784$ $\rightarrow 187230 \rightarrow 196590 \rightarrow 196591$	23
357887	$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow 18 \rightarrow 27 \rightarrow 45 \rightarrow 90$ $\rightarrow 180 \rightarrow 360 \rightarrow 720 \rightarrow 1440 \rightarrow 1485$ $\rightarrow 2970 \rightarrow 5940 \rightarrow 11880 \rightarrow 23760$ $\rightarrow 47520 \rightarrow 71280 \rightarrow 142560 \rightarrow 213840$ $\rightarrow 356400 \rightarrow 357885 \rightarrow 357887$	24
685951	$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 48$ $\rightarrow 96 \rightarrow 192 \rightarrow 384 \rightarrow 768 \rightarrow 769 \rightarrow 1538$ $\rightarrow 3076 \rightarrow 6152 \rightarrow 9228 \rightarrow 15380 \rightarrow 24608$ $\rightarrow 39988 \rightarrow 79976 \rightarrow 159952 \rightarrow 319904$ $\rightarrow 639808 \rightarrow 679796 \rightarrow 685948 \rightarrow 685951$	25
1176431	$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 10 \rightarrow 20 \rightarrow 40 \rightarrow 80$ $\rightarrow 160 \rightarrow 180 \rightarrow 340 \rightarrow 680 \rightarrow 1360$ $\rightarrow 2720 \rightarrow 4080 \rightarrow 8160 \rightarrow 16320 \rightarrow 32640$ $\rightarrow 48960 \rightarrow 97920 \rightarrow 97925 \rightarrow 195845$ $\rightarrow 391690 \rightarrow 587535 \rightarrow 1175070 \rightarrow 1176430$ $\rightarrow 1176431$	26

A straightforward, but rather expensive implementation of (24) can be carried out using the binary exponentiation method, requiring $n-1$ field squarings (S) and $n-2$ field multiplications (M), i.e.,

$$\mathrm{FLT}_{\mathrm{cost}}(n) = (n-1)S + (n-2)M. \tag{25}$$

Nevertheless, using an ingenious rearrangement of the required field operations it was shown in [25] that this calculation can be performed much more efficiently by using the so-called Itoh–Tsujii multiplicative inverse algorithm (ITMIA).

TABLE XII
SHORTEST ADDITION CHAINS FOR A SPECIAL CLASS OF EXPONENTS
(TABLE 3 OF 3)

exponent $e = c(r)$	Addition Chain	Length r
2211837	$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 10 \rightarrow 20 \rightarrow 40 \rightarrow 80 \rightarrow 160$ $\rightarrow 163 \rightarrow 326 \rightarrow 652 \rightarrow 1304 \rightarrow 2608$ $\rightarrow 5216 \rightarrow 10432 \rightarrow 20864 \rightarrow 41728$ $\rightarrow 83456 \rightarrow 166912 \rightarrow 166922 \rightarrow 333834$ $\rightarrow 500756 \rightarrow 1001512 \rightarrow 2003024$ $\rightarrow 2169946 \rightarrow 2211674 \rightarrow 2211837$	27
4169527	$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 48 \rightarrow 96 \rightarrow 192$ $\rightarrow 384 \rightarrow 768 \rightarrow 1536 \rightarrow 3072 \rightarrow 6144$ $\rightarrow 12288 \rightarrow 24576 \rightarrow 49152 \rightarrow 49344 \rightarrow 98688$ $\rightarrow 148032 \rightarrow 296064 \rightarrow 592128 \rightarrow 592129$ $\rightarrow 1184258 \rightarrow 2368516 \rightarrow 3552774 \rightarrow 4144903$ $\rightarrow 4169479 \rightarrow 4169527$	28
7624319	$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow 129$ $\rightarrow 258 \rightarrow 387 \rightarrow 774 \rightarrow 1548 \rightarrow 2322$ $\rightarrow 3870 \rightarrow 7740 \rightarrow 8127 \rightarrow 15867 \rightarrow 31734$ $\rightarrow 63468 \rightarrow 126936 \rightarrow 253872 \rightarrow 380808$ $\rightarrow 761616 \rightarrow 1523232 \rightarrow 3046464 \rightarrow 6092928$ $\rightarrow 7616160 \rightarrow 7624287 \rightarrow 7624319$	29
14143037	$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 72 \rightarrow 144$ $\rightarrow 216 \rightarrow 432 \rightarrow 864 \rightarrow 1728 \rightarrow 3456$ $\rightarrow 5184 \rightarrow 10368 \rightarrow 20736 \rightarrow 41472 \rightarrow 82944$ $\rightarrow 93312 \rightarrow 176256 \rightarrow 352512 \rightarrow 705024$ $\rightarrow 1410048 \rightarrow 2820096 \rightarrow 2820097 \rightarrow 2820313$ $\rightarrow 5640626 \rightarrow 8460939 \rightarrow 14101565 \rightarrow 14143037$	30

The ITMIA method is based on the observation that since $2^n - 2 = (2^{n-1} - 1) \cdot 2$, Fermat's little theorem identity can be rewritten as

$$A^{-1} \equiv A^{2^n-2} \equiv \left[A^{(2^{n-1}-1)}\right]^2. \tag{26}$$

Thereafter, ITMIA computes the field element $A^{2^{n-1}-1}$ using a recursive rearrangement of the finite field operations. It was shown in [9] and [36] that this algorithm requires $n-1$ field squarings plus only $l_{ac}(n-1)$ field multiplications, where $l_{ac}(n-1)$ is the length of the addition chain used to reach the number $n-1$. Therefore, the cost is given as

$$\mathrm{ITMIA}_{\mathrm{general}}(n) = (n-1)S + l_{ac}(n-1)M. \tag{27}$$

Comparing with (24), it can be noticed that although the number of field squarings required by the ITMIA method remains the same, the total number of multiplications N has been greatly reduced. Notice also that the concept of addition chains leads us to a natural way to generalize the Itoh–Tsujii algorithm reducing the number N even further.

TABLE XIII
OPTIMAL ADDITION CHAINS FOR $m = 32$ k. AIS = ARTIFICIAL IMMUNE SYSTEM (TABLE 1 OF 2)

$m-1$	AIS	AIS	[36]	[25]
31	$1 \to 2 \to 3 \to 6 \to 7 \to 14 \to 28 \to 31$	7	7	8
63	$1 \to 2 \to 3 \to 6 \to 7 \to 14 \to 28 \to 56 \to 63$	8	8	10
95	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 90 \to 95$	9	9	11
127	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 120 \to 126 \to 127$	10	10	12
159	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 144 \to 156 \to 159$	10	10	12
191	$1 \to 2 \to 4 \to 8 \to 16 \to 17 \to 34 \to 68 \to 136 \to 170 \to 187 \to 191$	11	11	13
223	$1 \to 2 \to 3 \to 6 \to 12 \to 13 \to 26 \to 52 \to 104 \to 208 \to 221 \to 223$	11	11	13
255	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 85 \to 170 \to 255$	10	10	14
287	$1 \to 2 \to 3 \to 5 \to 7 \to 14 \to 28 \to 56 \to 112 \to 224 \to 280 \to 287$	11	11	13
319	$1 \to 2 \to 3 \to 6 \to 12 \to 18 \to 36 \to 72 \to 144 \to 288 \to 306 \to 318 \to 319$	12	12	14
351	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 27 \to 54 \to 108 \to 216 \to 324 \to 351$	11	11	14
383	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 160 \to 320 \to 360 \to 380 \to 383$	12	13	15
415	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 83 \to 166 \to 332 \to 415$	11	12	14
447	$1 \to 2 \to 3 \to 6 \to 12 \to 18 \to 36 \to 72 \to 144 \to 288 \to 432 \to 444 \to 447$	12	12	15

TABLE XIV
OPTIMAL ADDITION CHAINS FOR $m = 32$ k. AIS = ARTIFICIAL IMMUNE SYSTEM (TABLE 2 OF 2)

$m-1$	AIS	AIS	[36]	[25]
479	$1 \to 2 \to 3 \to 6 \to 7 \to 14 \to 28 \to 56 \to 112 \to 224 \to 448 \to 476 \to 479$	12	13	15
511	$1 \to 2 \to 3 \to 5 \to 10 \to 15 \to 30 \to 60 \to 120 \to 240 \to 480 \to 510 \to 511$	12	12	16
575	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 23 \to 46 \to 92 \to 184 \to 368 \to 552 \to 575$	12	13	15
607	$1 \to 2 \to 3 \to 6 \to 12 \to 18 \to 36 \to 72 \to 144 \to 288 \to 576 \to 594 \to 606 \to 607$	13	13	15
639	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 26 \to 52 \to 104 \to 208 \to 416 \to 624 \to 636 \to 639$	13	13	16
767	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 83 \to 166 \to 332 \to 664 \to 747 \to 767$	13	14	17
799	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 192 \to 384 \to 768 \to 792 \to 798 \to 799$	13	13	15
863	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 43 \to 86 \to 172 \to 344 \to 688 \to 860 \to 863$	13	15	16
895	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 160 \to 163 \to 326 \to 652 \to 815 \to 895$	13	14	17

Since the original ITMIA method used a binary strategy, the number of field multiplications required by that algorithm is not optimal. Applying (3), the overall cost is then given as

$$\mathrm{ITMIA}_{\mathrm{binary}}(n) = (n-1)S + (H(n-1)-1)\,M \qquad (28)$$

where $H(n-1)$ is the Hamming weight of the binary representation of $n-1$. Takagi *et al.* [36] utilized a heuristic partially based on the factor method. They obtained shorter addition chains for $e = m-1$ than the ones generated by the ITMIA method, thus reducing the number of required multiplications of (28).

We compare the results obtained by our algorithm against the modified factor method presented by Takagi *et al.* [36] and the ITMIA binary method [25]. Tables XIII and XIV show the optimal addition chains for $m = 32k$ which is an important class of exponents for error-correcting code applications. The first column shows the target value, i.e., $e = m-1$. The addition chains found by the AIS algorithm and their respective lengths are listed in the second and in the third column, respectively. On a total of seven cases the AIS algorithm outperforms the method of [36], and in all cases considered, both algorithms outperform the ITMIA binary method.

As a second example, let us consider the family of exponents $e = p-1$, with p a prime number. This class of exponents is of special interest for elliptic curve cryptosystems defined over binary extension fields. For security reasons [24], that application utilizes the set of finite fields $F = \mathrm{GF}(2^n)$, with n being a

TABLE XV
OPTIMAL ADDITION CHAINS FOR $e = p - 1$, p A PRIME

$p-1$	AIS	AIS	ITMIA
162	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 64 \to 80 \to 81 \to 162$	9	9
166	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 83 \to 166$	9	10
172	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 43 \to 86 \to 172$	9	10
190	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 160 \to 180 \to 190$	10	12
192	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 192$	8	8
196	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 48 \to 49 \to 98 \to 196$	9	9
222	$1 \to 2 \to 3 \to 6 \to 12 \to 24 \to 48 \to 96 \to 192 \to 216 \to 222$	10	12
232	$1 \to 2 \to 4 \to 8 \to 16 \to 24 \to 28 \to 29 \to 58 \to 116 \to 232$	10	10
268	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 64 \to 66 \to 67 \to 134 \to 268$	10	10
270	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 90 \to 180 \to 270$	10	11
292	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 64 \to 72 \to 73 \to 146 \to 292$	10	10
330	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 160 \to 320 \to 330$	10	11
378	$1 \to 2 \to 4 \to 8 \to 16 \to 18 \to 36 \to 72 \to 144 \to 288 \to 360 \to 378$	11	13
382	$1 \to 2 \to 3 \to 5 \to 10 \to 20 \to 40 \to 80 \to 160 \to 320 \to 360 \to 380 \to 382$	12	14
388	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 64 \to 96 \to 97 \to 194 \to 388$	10	10
442	$1 \to 2 \to 3 \to 6 \to 12 \to 13 \to 26 \to 52 \to 104 \to 208 \to 416 \to 442$	11	13
462	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 33 \to 66 \to 132 \to 264 \to 396 \to 462$	11	13
490	$1 \to 2 \to 3 \to 5$ 95 $\to 10 \to 20 \to 40 \to 80 \to 160 \to 320 \to 480 \to 490$	11	13
508	$1 \to 2 \to 3 \to 6 \to 12 \to 14 \to 28 \to 30 \to 60 \to 120 \to 240 \to 480 \to 508$	12	14
520	$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 64 \to 65 \to 130 \to 260 \to 520$	10	10

prime in the range [160, 521]. Table XV summarizes the results obtained by the AIS heuristic and the binary method. In 12 out of 20 of the cases considered, the AIS algorithm obtains better results than the ITMIA binary method, and is no worse in the other cases.

In order to quantify the solution's quality obtained from the addition-chain-based ITMIA method, let us consider the computation of multiplicative inverses over the finite field $F = \mathrm{GF}(2^{509})$, by using Fermat's identity, i.e., $A^{-1} = A^{2^{509}-2}$.

By consulting the second to last entry of Table XV, namely, $p - 1 = 508$, we see that its corresponding shortest addition chain (as it was found by the AIS heuristic), has length 12. Therefore, according to (27), the required number of arithmetic operations for this 509-bit exponent is given as

$$\begin{aligned}\mathrm{ITMIA}_{\mathrm{cost}}(n = 509) &= (n-1)S + l_{ac}(n-1)M \\ &= 508S + 12M.\end{aligned}$$

Using the ratio $\#Operations/n$ as a figure of merit, we get

$$\frac{\mathrm{ITMIA}_{\mathrm{cost}}(n = 509)}{n} = 1.023 \qquad (29)$$

which according with the lower bound (9), is about the best cost that one can expect from an exponentiation computation.

B. AIS Heuristic Combined With the Sliding Window Method

Perhaps the single most important arithmetic operation for public-key cryptography is exponentiation. The RSA encryption/decryption and signing/verification schemes are based on the computation of an exponentiation operation, namely, $M^e \bmod n$, where e is a fixed number, M is an arbitrarily chosen numeric message, and n the product of two large primes $n = pq$. Additionally, the Diffie–Hellman key exchange scheme the ElGamal signature scheme and the digital signature standard (DSS) also require the computation of modular exponentiation [4], [27], [32].

The exponentiation methods described in this paper are all focused on the so-called fixed-exponent exponentiation problem, i.e., the exponent e is fixed and arbitrary choices of the base M are allowed. RSA encryption and decryption schemes are based on these kind of algorithms.

Since e is a fixed number, we can compute its addition chain in an *offline* fashion. Therefore, under this scenario, the computational time needed for computing the optimal addition chain becomes a noncritical design issue. Usually, we will precompute that addition chain well before the beginning of the real field exponentiation computation.

Fig. 15 shows the customary figure of merit $\#Operations/m$, i.e., the average number of operations divided by the total number of bits m, for the m-ary, and the AIS sliding window algorithms as a function of $m = 128$, 256, 512, 1024. Those exponent lengths are regularly used in cryptographic applications.

Table XVI compares the performance of the traditional sliding window method (as reported in [27]) against the sliding window method combined with the AIS heuristic. Those two methods were applied on exponents e with relatively large bit-length m, namely, $m = 128$, 256, 512, 1024. The AIS sliding window method was tested allowing arbitrarily large MSWs candidates but fixing the maximum size allowed for all the other nonzero windows to a value $k \in [6, 7]$. We also fixed

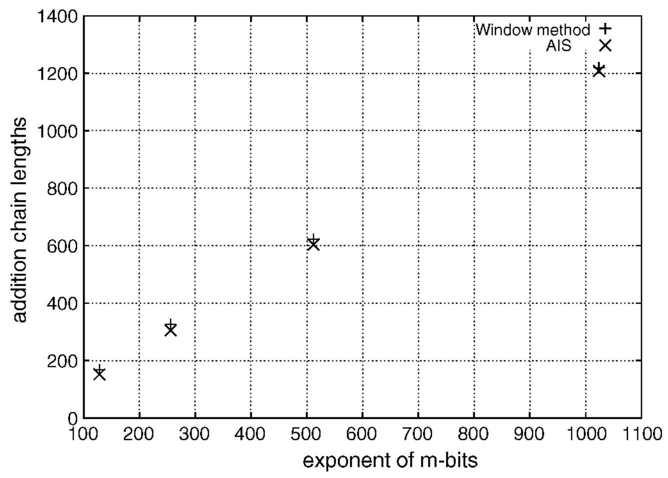

Fig. 15. AIS sliding window method against the sliding window method.

TABLE XVI
PERFORMANCE OF THE AIS METHOD FOR LARGE EXPONENTS

m	*Sliding window Method [27]*		*AIS Heuristic*		
	length	k	length	MSW size	q
128	156	4	152	17	2
256	308	4	304	13	2
512	607	5	604	11	2
1024	1195	5	1196	6	5

the maximum value of consecutive zeros to $q = 2$, except for the case $m = 1024$, where $q = 5$ was used.

As it can be seen in Table XVI, our strategy outperforms the window method for the first three cases, namely, $m = 128, 256$, 512. However, the AIS strategy tends to deteriorate its performance as the bit length grows larger. In the case of $m = 1024$, the traditional sliding window method shows a slightly better performance than the AIS strategy.

VIII. CONCLUSION

In this paper, we presented an AIS heuristic applied to the problem of finding optimal addition chains for field exponentiation computations. We only emulated some immunological actors and mechanisms, namely, antibodies and antigens, hypermutation, cloning, and secondary response. By doing so, we believe that we were able to confection an algorithm that is conceptually simple but at the same time effective and efficient.

The AIS heuristic proposed in this research work was capable of finding almost all the optimal addition chains for any given fixed exponent e with $e < 4096$, exhibiting a high success rate of 99.6%. Furthermore, in order to assess the actual power of the AIS strategy as a search engine, we used it for generating the shortest addition chains of a class of exponents particularly hard to optimize. In all cases considered, the AIS strategy was able to find the optimal values.

Additionally, we collected the associated uncertainty of our results through the computation of the experiments' confidence intervals. This was done by applying a bootstrap resampling statistical test. The importance of performing this type of test lies in the fact that only by using statistical tests can one reasonably ensure that the results yielded by a probabilistic heuristic are consistent and independent of the random seed used. This way, we provided statistical evidence that the experimental lower and upper average values are very close to each other. Thus, it is fair to say that the average algorithm behavior is quite similar from one execution to the other, which is a desirable feature for a probabilistic heuristic to exhibit.

As a means to show how the concept of a powerful heuristic for finding addition chains could be applied in practice, we included two code-theory applications.

The first application consisted on utilizing the AIS strategy in the problem of finding optimal addition chains for field exponentiation computations over binary extension fields. The results obtained by our scheme yielded some of the shortest reported lengths for exponents typically used when computing field multiplicative inverses for error-correcting and elliptic curve cryptographic applications.

The second application consisted of developing a strategy that combined the sliding window method with the AIS-based heuristic. While, in general, optimal solutions for exponents with large bit lengths are unknown, we provided a comparison of our experimental results against the ones obtained by the traditional sliding window method. Our experiments show that the AIS strategy tends to be better for moderated sizes of $e \in [128, 256, 512]$. However, for larger sizes, the AIS strategy is not as efficient as the traditional sliding window.

Future work includes improving the performance of our strategy for both, exponents with moderated size (i.e., 32-bit length or less); and when dealing with extremely large exponents, as the ones typically used in RSA and DSA cryptosystems. We are also planning to explore the performance of other biologically inspired heuristics when applied to the optimal addition chain problem.

ACKNOWLEDGMENT

The authors gratefully acknowledge the comments from the anonymous reviewers, which greatly helped them to improve the contents of the paper.

REFERENCES

[1] A. Schönhage, "A lower bound for the length of addition chains," *Theor. Comput. Sci.*, vol. 1, pp. 1–12, 1975.
[2] U. Aickelin, J. Greensmith, and J. Twycross, "Immune system approaches to intrusion detection—A review," in *Proc. 3rd Int. Conf. Artif. Immune Syst., ICARIS 2004*, G. Nicosia, V. Cutello, P. J. Bentley, and J. Timmis, Eds., Catania, Sicily, Italy, Sep. 2004, vol. 3239, Lecture Notes in Computer Science, pp. 316–329.
[3] P. S. Andrews and J. Timmis, "Inspiration for the next generation of artificial immune systems," in *Proc. 4th Int. Conf. Artif. Immune Syst., ICARIS 2005*, C. Jacob, M. L. Pilat, P. J. Bentley, and J. Timmis, Eds., Aug. 2005, vol. 3627, Lecture Notes in Computer Science, pp. 126–138.
[4] *National Standards for Financial Institution Key Management (Wholesale)*, ANSI X9.17 (Revised), American Bankers Assoc., Washington, DC, 1986.
[5] F. Bergeron, J. Berstel, and S. Brlek, "Efficient computation of addition chains," *J. de théorie des nombres de Bordeaux*, vol. 6, pp. 21–38, 1994.
[6] J. Bos and M. Coster, "Addition chain heuristics," in *Proc. Adv. Cryptology, CRYPTO 89*, G. Brassard, Ed., 1989, vol. 435, Lecture Notes in Computer Science, pp. 400–407.

[7] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson, "Fast exponentiation with precomputation," in *Proc. Adv. Cryptology, EUROCRYPT 92*, R. A. Rueppel, Ed., 1992, vol. 658, Lecture Notes in Computer Science, pp. 200–207.

[8] F. M. Burnet, "Clonal selection and after," in *Theoretical Immunology*, G. I. Bell, A. S. Perelson, and G. H. Pimgley, Jr., Eds. New York: Marcel Dekker, 1978, pp. 63–85.

[9] N. Cruz-Cortes, F. Rodriguez-Henriquez, and C. Coello Coello, "On the optimal computation of finite field exponentiation," in *Proc. 9th Ibero-Amer. Conf. AI Adv. Artif. Intell., IBERAMIA 2004*, C. Lemaître, C. Reyes, and J. González, Eds., Nov. 2004, vol. 3315, Lecture Notes in Computer Science, pp. 747–756.

[10] V. Cutello and G. Nicosia, "An immunological approach to combinatorial optimization problems," in *Proc. Adv. Artif. Intell., IBERAMIA 2002*, Seville, Spain, Nov. 2002, vol. 2527, Lecture Notes in Artificial Intelligence, pp. 361–370.

[11] D. Bleinchenbacher and A. Flammenkamp, "An efficient algorithm for computing shortest addition chains," 1997. [Online]. Available: http://www.uni-bielefeld.de/~achim

[12] D. Dasgupta and N. Attoh-Okine, "Immunity-based systems: A survey," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Orlando, FL, Oct. 12–15, 1997, pp. 369 –374.

[13] D. Dasgupta, K. KrishnaKumar, D. Wong, and M. Berry, "Negative selection algorithm for aircraft fault detection," in *Proc. 3rd Int. Conf. Artif. Immune Syst., ICARIS 2004*, G. Nicosia, V. Cutello, P. Bentley, and J. Timmis, Eds., Sep. 2004, pp. 13–16.

[14] L. Nunes de Castro and J. Timmis*, An Introduction to Artificial Immune Systems: A New Computational Intelligence Paradigm*. Berlin, Germany: Springer-Verlag, 2002.

[15] L. Nunes de Castro and F. J. Von Zuben, "Learning and optimization using the clonal selection principle," *IEEE Trans. Evol. Comput.*, vol. 6, no. 3, pp. 239–251, Jun. 2002.

[16] S. Forrest and S. A. Hofmeyr, "Immunology as information processing," in *Design Principles for the Immune System and Other Distributed Autonomous Systems,* Santa Fe Institute Studies in the Sciences of Complexity, L. A. Segel and I. Cohen, Eds. Oxford, U.K.: Oxford Univ. Press, 2000, pp. 361–387.

[17] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. IEEE Symp. Comput. Security Privacy*, 1996, pp. 120–128.

[18] S. A. Frank, "The design of natural and artificial adaptive systems," in *Adaptation*, R. Michael and L. George, Eds. New York: Academic Press, 1996, ch. 12, pp. 451–505.

[19] F. González and D. Dasgupta, "Anomaly detection using real-valued negative selection," *Genetic Programming Evolv. Mach.*, vol. 4, no. 4, pp. 383–403, Dec. 2003.

[20] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, no. 1, pp. 129–146, Apr. 1998.

[21] P. Hajela and J. Lee, "Constrained genetic search via schema adaptation. An immune network solution," in *Proc. 1st World Congr. Stuctural Multidisciplinary Opt.*, N. Olhoff and G. I. N. Rozvany, Eds., Goslar, Germany, 1995, pp. 915–920.

[22] P. K. Harmer, P. D. Williams, G. H. Gunsch, and G. B. Lamont, "An artificial immune system architecture for computer security applications," *IEEE Trans. Evol. Comput.*, vol. 6, no. 3, pp. 252–280, Jun. 2002.

[23] E. Hart, "The evolution and analysis of a potential antibody library for use in job-shop scheduling," in *New Ideas in Optimization*, D. C. Dorigo and F. Glover, Eds. New York: McGraw-Hill, 1999, pp. 185–202.

[24] *Standard Specifications for Public-Key Cryptography, Draft Version D18*, IEEE P1363 (IEEE standards documents), Nov. 2004. [Online]. Available: http://grouper.ieee.org/groups/1363/

[25] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in $\mathrm{GF}(2^m)$ using normal basis," *Inf. Comput.*, vol. 78, pp. 171–177, 1988.

[26] D. E. Knuth*, The Art of Computer Programming—3rd. ed*. Reading, MA: Addison-Wesley, 1997.

[27] Ç. K. Koç, High-speed RSA implementation RSA Laboratories, Redwood City, CA, Tech. Rep. TR 201, 1994, 71 pages.

[28] Ç. K. Koç, "Analysis of sliding window techniques for exponentiation," *Computer and Mathematics With Applications*, vol. 30, no. 10, pp. 17–24, Oct. 1995.

[29] N. Kunihiro and H. Yamamoto, "Window and extended window methods for addition chain and addition-subtraction chain," *IEICE Trans. Fundamentals*, vol. E81-A, no. 1, pp. 72–81, Jan. 1998.

[30] N. Kunihiro and H. Yamamoto, "New methods for generating short addition chains," *IEICE Trans. Fundamentals*, vol. E83-A, no. 1, pp. 60–67, Jan. 2000.

[31] G. B. Lamont, R. E. Marmelstein, and D. A. Van Veldhuizen, "A distributed architechture for a self-adaptive computer virus immune system," in *New Ideas in Optimization*. New York: McGraw-Hill, 1999, pp. 167–183.

[32] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone*, Handbook of Applied Cryptography*. Boca Raton, FL: CRC, 1996.

[33] N. Nedjah and L. D. Mourelle, "Efficient preprocessing for large window-based modular exponentiation using genetic algorithms," *Developments in Applied Artificial Intelligence*, vol. 2718, Lecture Notes in Artificial Intelligence, pp. 625–635, 2003.

[34] J. Olivos, "On vectorial addition chains," *J. Algorithms*, vol. 2, no. 1, pp. 13–21, 1981.

[35] S. Stepney, R. E. Smith, J. Timmis, and A. M. Tyrrell, "Towards a conceptual framework for artificial immune systems," in *Proc. 3rd Int. Conf. Artif. Immune Syst., ICARIS 2004*, G. Nicosia, V. Cutello, P. Bentley, and J. Timmis, Eds., Sep. 2004, vol. 3239, Lecture Notes in Computer Science, pp. 53–64.

[36] N. Takagi, J. Yoshiki, and K. Tagaki, "A fast algorithm for multiplicative inversion in $\mathrm{GF}(2^m)$ using normal basis," *IEEE Trans. Comput.*, vol. 50, no. 5, pp. 394–398, May 2001.

[37] J. von zur Gathen and M. Nöcker, "Computing special powers in finite fields: Extended abstract," in *Proc. International Symp. Symbolic and Algebraic Comput.*, 1999, pp. 83–90.

[38] Y. Lee, H. Kim, S. Hong, and H. Yoon, "Expansion of sliding window method for finding shorter addition/subtraction-chains," *Int. J. Network Security* vol. 2, no. 1, pp. 34–40, Jan. 2006. [Online]. Available: http://isrc.nchu.edu.tw/ijns/

[39] Y. Tsuruoka and K. Koyama, "Fast computation over elliptic curves $E(F_{q^n})$ based on optimal addition sequences," *IEICE Trans. Fundamentals*, vol. E84-A, no. 1, pp. 114–119, Jan. 2001.

[40] Y. Yacobi, "Exponentiating faster with addition chains," in *Proc. Adv. Cryptology, EUROCRYPT 90*, I. B. Damgard, Ed., 1990, vol. 473, Lecture Notes in Computer Science, pp. 222–229.

[41] S. M. Yen, "Improved normal basis inversion in $\mathrm{GF}(2^m)$," *IEE Electronic Lett.*, vol. 33, no. 3, pp. 196–197, Jan. 1997.

Nareli Cruz-Cortés received the B.Sc. degree in Computer Engineering from the Technological Institute of Tepic, Nayarit, México, in 1995, the M.Sc. degree in artificial intelligence from the University of Veracruz and LANIA, Veracruz, México, in 2000, and the Ph.D. degree in electrical and computer engineering from CINVESTAV, México City, México, in 2004.

Currently, she is a Lecturer at the Center for Computing Research, National Polytechnic Institute, México. Her major research interests are in combinatorial and multiobjective optimization, genetic algorithms, and artificial immune systems.

Francisco Rodríguez-Henríquez (M'03) received the B.Sc. degree in electrical engineering from the University of Puebla, Puebla, México, in 1989, the M.Sc. degree in electrical and computer engineering from the National Institute of Astrophysics, Optics and Electronics (INAOE), Puebla, in 1992, and the Ph.D. degree in electrical and computer engineering from Oregon State University, Corvallis, in 2000.

Currently, he is a Professor (CINVESTAV-3B Researcher) with the Department of Computer Science, CINVESTAV-IPN, Mexico City, México, which he joined in 2002. He is an Alumni Member and Research Associate of the Information Security Laboratory, Oregon State University. His major research interests are in data security, cryptography, finite fields, error correcting codes, and mobile computing.

Carlos A. Coello Coello (M'89–SM'04) received the B.Sc. degree in civil engineering from the Universidad Autónoma de Chiapas, México, in 1991, and the M.Sc. and Ph.D. degrees in computer science from Tulane University, New Orleans, LA, in 1993 and 1996, respectively.

He is currently a Professor (CINVESTAV-3D Researcher) with the Department of Computer Science, CINVESTAV-IPN, Mexico City, México. He has authored and coauthored over 120 technical papers and several book chapters. He has also coauthored the book *Evolutionary Algorithms for Solving Multi-Objective Problems* (Kluwer, 2002) and has coedited the book *Applications of Multi-Objective Evolutionary Algorithms* (World Scientific, 2004). His major research interests are: evolutionary multiobjective optimization, constraint-handling techniques for evolutionary algorithms, and evolvable hardware.

Dr. Coello is a member of the Association for Computing Machinery (ACM), Sigma Xi, and the Mexican Academy of Sciences. He has served on the program committees of over 40 international conferences and has been Technical Reviewer for over 30 international journals including the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION in which he also serves as Associate Editor. He is a member of the editorial boards of the journals *Evolutionary Computation*, *Engineering Optimization*, and *Soft Computing*. He also chairs the *Task Force on Multi-Objective Evolutionary Algorithms* of the IEEE Computational Intelligence Society.