LETTER

An Efficient Exponentiation Algorithm in $GF(2^m)$ Using Euclidean Inversion

Wei HE[†a)], *Nonmember*, Yu ZHANG[†], *Member*, *and* Yin LI[††], *Nonmember*

SUMMARY We introduce a new type of exponentiation algorithm in $GF(2^m)$ using Euclidean inversion. Our approach is based on the fact that Euclidean inversion cost much less logic gates than ordinary multiplication in $GF(2^m)$. By applying signed binary form of the exponent instead of classic binary form, the proposed algorithm can reduce the number of operations further compared with the classic algorithms.

key words: *finite field, exponentiation, Euclidean inversion, cryptography*

1. Introduction

Finite field $GF(2^m)$ has many practical applications in public key cryptography or error correcting codes. Therefore, it is crucial to perform the underlying arithmetic operations, including multiplication, exponentiation and inversion, efficiently. Generally speaking, exponentiation and inversion are much more time-consuming than other arithmetic operations. Therefore, many researchers devoted particular interest to provide efficient implementations for these operations. So far, there are three main strategies for computing the inversion over $GF(2^m)$, i.e., Eulerian (Euler-Fermat theorem) based, Gaussian based and Euclidean based algorithms. All these approaches and their variations have been extensive studied in [5]. Compared with the inversion, the exponentiation $\alpha^E (E > 0)$ in $GF(2^m)$ is even harder, as the exponent can be any nonzero numbers. In fact, according to Euler-Fermat theorem, the inverse of $\alpha \in GF(2^m)$ can be rewritten as $\alpha^{-1} = \alpha^{2^m-2}$, which indicates that inversion could be recognized as a special case of exponentiation with related exponent fixed. The main approaches for computing exponentiation consist of binary method [2], [3], sliding window [3], addition chain [4] or their variations [6]. Algorithm 1 presents the well-known binary method for exponentiation.

It is obvious that binary exponentiation costs at most $n - 1$ field multiplications, where n represents the number of bits in the binary form of E. The sliding window is an extension of the binary method, which partition the exponent into zero and nonzero parts. If these small exponentiations related to nonzero parts are pre-computed and stored, the overall exponentiation will obtain a faster implementation. Addition chain [2] can also be applied to calculate exponentiation. The key idea of this approach builds an addition chain from 1 to E, which corresponds to the intermediate steps of exponentiation. Theoretically, it costs the fewest multiplications if the related shortest addition chain is found. However, different exponents always lead to different addition chain, which requires updating related addition chain frequently. In [4], the authors proposed sliding window methods for exponentiation under an addition-subtraction chain (an extension of addition chain). The authors in [6] combined the idea of sliding window and addition chain to obtain the optimal window size.

Algorithm 1 Binary Exponentiation [2]

```
Input: $A \in GF(2^m), f(x), e = (e_{n-1}e_{n-2} \cdots e_1e_0)_2$
Output: $B = A^e \bmod f(x)$
1: $B := A$;
2: for $i$ from $n-2$ to 0 do
3:     $B := B^2 \bmod f(x)$;
4:     if $e_i = 1$ then
5:         $B = B \cdot A \bmod f(x)$;
6:     end if
7: end for
8: return $B$
```

Motivation. In this letter, our work is devote to building an efficient exponentiation scheme to reduce the number of multiplications further, but maintain a relatively simple structure. Our approach is based on the fact that $\alpha^{2^t-1} = \alpha^{2^t} \cdot \alpha^{-1}$. A straightforward way to compute such a exponentiation costs $t-2$ field multiplications (using binary algorithm) plus $t-1$ squarings. On the contrary, this exponent can also be obtained using one multiplication, one inversion plus t squarings. Obviously, if the inverse of α can be easily obtained, the latter approach is much more efficient than the former one, as squarings are usually easy to calculate compared with multiplication.

The authors in [5] already demonstrated that the inversion using Euclidean algorithm costs only m XOR gates with $2m\,T_X$ delays, where T_X is the delay of one 2-input XOR gate. That is to say, Euclidean inversion is even faster than ordinary field multiplication in $GF(2^m)$ (see Table 1). Based on this observation, an efficient exponentiation algorithm is developed in the following sections.

2. Signed Binary Form

In order to utilize inversion, we consider an alternative rep-

Manuscript received January 12, 2022.
Manuscript revised March 25, 2022.
Manuscript publicized April 26, 2022.

[†]The authors are with Department of Computer Science and Technology, Xinyang Normal University, Henan, China.
[††]The author is with School of Cyberspace Security, Dongguan University of Technology, Guangdong, China.
a) E-mail: violahw@126.com

DOI: 10.1587/transfun.2022EAL2002

Table 1 Comparison of Euclidean inversion and ordinary multiplication over $GF(2^m)$ generated with $x^m + x^k + 1$ [5].

	$\#AND$	$\#XOR$	Delay
Mult.	m^2	$m^2 - 1$	$\leq T_A + \lceil(2+\log_2(m-1))\rceil T_X$
Euclidean	0	$2m$	$2mT_X$
T_A represents the delay of an 2-input AND gate			

resentation mentioned in [4], i.e., signed binary form, for the exponent E other than the binary form. The signed binary form consists of the symbols “1”, “0” and “−1”. For concision, we use the notation $\bar{1} = -1$ instead of “−1”. For example, 15 is “1111” in binary form, and is “$1000\bar{1}$” in signed binary form. Also note that the signed binary form w.r.t. a fixed number is not unique. For instance, $23 = 10111 = 1100\bar{1} = 10\bar{1}00\bar{1}$.

Although the canonical signed binary representation (CSBR) generated by Booth algorithm [1] are sparse and minimal, we do not use it for efficiency consideration. The reason is that, if there is only two consecutive “1”s in the binary form of E, the signed binary form cannot optimize its exponentiation. Let us consider the simplest case $\alpha^3 = \alpha^4 \cdot \alpha^{-1}$. The direct computation of α^3 only cost one field multiplications plus one squaring, while $\alpha^4 \cdot \alpha^{-1}$ cost one multiplication, two squarings as well as one inversion. Only if there exist at least three consecutive “1”s in the binary form, our method is better than the binary exponentiation. Accordingly, the following algorithm gives the converting algorithm from binary to signed binary form.

Algorithm 2 Signed Binary Form Conversion

```
Input: Binary form of $E = e_{n-1}e_{n-2}\cdots e_1e_0$
Output: Signed binary form of $E = (e'_n e'_{n-1}\cdots e'_1 e'_0)_{sig2}$
 1: for $i$ from 0 up to $n-3$ do
 2:     $e'_i := e_i$;
 3:     if $e_i = e_{i+1} = e_{i+2} = 1$ then
 4:         $j := i + 3$;
 5:         while $e_j \neq 0$ do
 6:             $j := j + 1$;
 7:         end while
 8:         for $k$ from $i+1$ up to $j$ do
 9:             $e'_k := 0$;
10:         end for
11:         $e'_{j+1} := 1, i := j + 1$;
12:     end if
13: end for
14: return $e'_n e'_{n-1}\cdots e'_1 e'_0$
```

Description: The notation $(*)_{sig2}$ represent the signed binary form of E. Algorithm 2 scans the binary form of E from left to right. If there is no at least three consecutive “1”s in the binary form of E, the binary form keeps unchanged, otherwise it transforms all consecutive “1”s of the form “$11\cdots11$” to “$100\cdots0\bar{1}$”. Accordingly, we prefer the signed form of 23 as $1100\bar{1}$ rather than $10\bar{1}00\bar{1}$.

Proposition 1. *The signed binary form of E contains at most one more bit compared with its binary form.*

Proof. Recall that the binary form of E is $e_{n-1}e_{n-2}\cdots e_0$. We first know that the most significant bit of E is certainly 1, i.e., $e_{n-1} = 1$. Then, the value of e_{n-2} can be 1 or 0. If $e_{n-2} = 0$, no matter which values the rest bits $e_{n-3}\cdots e_1e_0$ are, the conversion from binary to signed binary form can only set e'_{n-2} to 1, and $e'_{n-1} = e_{n-1}$. Thus, in this case, the number of bits in signed binary form of E is equal to that of the binary form of E.

If $e_{n-2} = 1$ and $e_{n-3} = 0$, one can easily check that the signed binary form of E also contains n bits, which is similar to previous case.

If $e_{n-2} = e_{n-3} = e_{n-1} = 1$, according to the transformation rule, there are at least three consecutive “1”s, thus, the signed binary form of E set $e'_{n-1}, e'_{n-2}, e'_{n-3}$ to 0 and one extra bit e_n to 1. In this case, the signed binary form of E has $n+1$ bits, which contains one more bits compared with its binary form. □

3. Exponentiation Algorithm Using Euclidean Inversion

We now give the explicit exponentiation algorithm using signed binary form. Denoted by $EU(*)$ the Euclidean inversion computation. Then, the proposed exponentiation algorithm is given as follows:

Algorithm 3 Signed Binary Exponentiation using Inversion

```
Input: $A \in GF(2^m), f(x), e = (e_{n-1}e_{n-2}\cdots e_1e_0)_{sig2}$
Output: $B = A^e \bmod f(x)$
 1: $B := A$;
 2: for $i$ from $n-2$ to 0 do
 3:     $B := B^2 \bmod f(x)$;
 4:     if $e_i = 1$ then
 5:         $B := B \cdot A \bmod f(x)$;
 6:     else
 7:         if $e_i = \bar{1}$ then
 8:             $B := B \cdot EU(A) \bmod f(x)$;
 9:         end if
10:     end if
11: end for
12: return $B$
```

Compared with Algorithm 1, Algorithm 3 only requires one more component for Euclidean inversion. In hardware implementation, based on Table 1, one can check that only $2m$ more XOR gates are required.

Complexity Analysis. Assume that the signed binary form of the exponent contains $\alpha(\alpha \geq 0)$ “$\bar{1}$”s and $\beta(\beta \geq 0)$ “1”s. From Algorithm 3, it is easy to see that the proposed algorithm costs $n-2$ squarings and α inversions and $\alpha+\beta$ field multiplications. As mentioned in Sect. 2, since we only convert the exponent that contains at least three consecutive “1”s into its signed binary form, at least one multiplication can be saved at the cost of one Euclidean inversion. Now we investigate the number of squarings required in Algorithm 3. We have the following proposition.

Table 2 Computation of A^{23}.

i	Rule	Result
3	$B = A^2 \cdot A$	A^3
2	$B = B^2$	A^6
1	$B = B^2$	A^{12}
0	$B = B^2 \cdot A^{-1}$	A^{23}

Proposition 2. *Algorithm 3 costs at most one more squaring compared with Algorithm 1.*

Proof. Based on Algorithms 1 and 3, it is clear that the numbers of required squaring in these algorithms are equal to $n-1$, where n is the number of bits of e, in both binary or signed binary form.

Then we consider number of bits for different representations. Based on proposition 1, we know that signed binary form has at most one more bit compared with the binary one. We then directly conclude the proposition. □

An Example. Consider the exponentiation of A^{23} over the field $GF(2^{27})$, where A is an arbitrary element of $GF(2^{27})$. Based on previous description, the exponent 23 is rewritten in signed binary form $23 = 1100\bar{1}$. Then, according to Algorithm 3, we can compute the exponentiation as follows: Obviously, in this example, our algorithm only costs two multiplications, one inversion and four squarings. On the contrary, binary algorithm costs three multiplications plus four squarings.

Comparison. Please note that both the sliding window and the addition chain approaches require pre-computations, which are more suitable for software implementation. In the following, we make an explicit comparison between the binary exponentiation algorithm and our proposal. Firstly, as shown in Sect. 3, our scheme costs $2m$ more XOR gates than the binary exponentiation, by adding a Euclidean inversion component. However, with the results presented in this paper, we argue that our proposal can be even more efficient than the binary exponentiation algorithm in terms of bit operations. It is worthy to note that the number of bit operations can also be used to evaluate the algorithm efficiency [7], where more bit operations lead to more signal processing, more energy consumption.

Since the numbers of required bit operations depend on the representation of the exponent E, it is difficult to give an explicit complexity formulation w.r.t all the exponents. For simplicity, we assume that the signed binary form of E has one more bit than its binary form (n bits), but the signed binary form is more sparse. However, if there exist no at least three consecutive 1s in the binary form of E, we do not use the signed binary form as it leads to no reduction of the number of multiplications. In fact, we have searched all the exponents $E \in [1, 2^{20}]$ and found that more than 78% of such exponents contain at least three consecutive "1"s, the proportion also increases with the increase of E. Thus, our assumption here is reasonable. Denoted by $h(E)$ the Hamming weight of binary form of E. So, the Hamming weight of its signed binary form is no more than $h(E) - 1$. Also provide that $GF(2^m)$ is generated with an irreducible trinomial and the polynomial basis is used. If there is no irreducible trinomial for a certain degree, an irreducible pentanomial can be used, but the corresponding multiplier has higher space and time complexities.

Table 3 Comparison of bit operations for Algorithm 1 and 3.

Approaches	Number of bit operations
Algorithm 1	$2h(E) \cdot m^2 + mn - 4m^2 - h(E) - m + 2$
Algorithm 3	$2h(E) \cdot m^2 + 2\alpha m - 6m^2 + mn - h(E) + 3$

We utilize the squarer (cost at most m XOR gates), the field multiplier presented in [7], and the Euclidean Algorithm presented in [5] for hardware implementation of Algorithms 1 or 3, please see Table 1. For the exponentiation A^E in $GF(2^m)$, Algorithm 1 requires $h(E)-2$ field multiplications and $n-1$ squarings. Meanwhile, our proposal uses at most $h(E) - 3$ field multiplications, n squarings plus α inversions, where α represents the number of $\bar{1}$ in the signed binary form of E. The following table presents bit operations for two algorithms.

Since $\alpha << m$, it is obvious that, even if our proposal saves one field multiplication, it still requires fewer bit operations than the classic binary exponentiation. In [8], the authors defined new Modular Exponential (MODP) Groups for the Internet Key Exchange (IKE) protocol. They gave 1536, 2048, 3072, 4096, 6144 and 8192 bit prime numbers for Diffie-Hellman groups. The binary extension fields of the same security level are $GF(2^{1536}), GF(2^{2048}), GF(2^{3072}), GF(2^{4096}), GF(2^{6144})$ and $GF(2^{8192})$. In these cases, the degrees of the generating polynomials are very big, we can easily check that, even if our scheme only reduces one field multiplication, a lot of bit operations can be saved and our algorithm is more favourable.

4. Conclusion

In this paper, we have proposed a new $GF(2^m)$ exponentiation algorithm using Euclidean inversion. By choosing a signed binary form of the exponent, the proposed scheme can reduce the number of field multiplications further by substituting consecutive multiplications with one multiplication plus a inversion. Compared with classic binary exponentiation, our proposal only costs m more XOR gates but has fewer bit operations for certain exponent, which is more efficient in practical implementation. We also demonstrate that such exponent is abundant.

Acknowledgments

This work has been supported by the National Natural Science Foundation of China (Grant No. 61601396).

References

[1] K. Hwang, Computer Arithmetic, Wiley, New York, 1979.

[2] D. Knuth, The Art of Computer Programming, Addison-Wesley, Stanford, CA, 1998.

[3] C.K. Koç, "Analysis of sliding window techniques for exponentiation," Comput. Math. Appl., vol.30, no.10, pp.17–24, 1995.

[4] 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.

[5] M. Leone and M. Elia, "On the complexity of parallel algorithms for computing inverses in $GF(2^m)$ with m prime," Acta Appl. Math., vol.93, no.1-3, pp.149–160, 2006.

[6] A.M. Noma, A. Muhammed, Z.A. Zukarnain, M.A. Mohamed, and D. Pham, "Iterative sliding window method for shorter number of operations in modular exponentiation and scalar multiplication," Cogent Engineering, vol.4, no.1, 1304499, 2017.

[7] H. Wu, "Bit-parallel finite field multiplier and squarer using polynomial basis," IEEE Trans. Comput., vol.51, no.7, pp.750–758, 2002.

[8] T. Kivinen and M. Kojo, "More modular exponential (MODP) Diffie–Hellman groups for internet key exchange (IKE)," Internet Engineering, Task Force 2003, RFC 3526 (Standards Track).