

Accelerating exp-log based finite field multiplication

Nicolas Belleville

▶ **To cite this version:**

Nicolas Belleville. Accelerating exp-log based finite field multiplication. Cryptology ePrint Archive, 2023, 2023, pp.375. cea-04094208

HAL Id: cea-04094208
https://cea.hal.science/cea-04094208

Submitted on 10 May 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Accelerating exp-log based finite field multiplication

Nicolas Belleville
nicolas.belleville@cea.fr
Univ. Grenoble Alpes, CEA, List, F-38000 Grenoble, France

Abstract

Finite field multiplication is widely used for masking countermeasures against side-channel attacks. The execution time of finite field multiplication implementation is critical as it greatly impacts the overhead of the countermeasure. In this context, the use of exp-log tables is popular for the implementation of finite field multiplication. Yet, its performance is affected by the need for particular code to handle the case where one of the operands equals zero, as log is undefined for zero. As noticed by two recent papers, the zero case can be managed without any extra code by extending the exp table and setting $log[0]$ to a large-enough value. The multiplication of a and b then becomes as simple as: $exp[log[a] + log[b]]$. In this paper, we compare this approach with other implementations of finite field multiplication and show that it provides a good trade-off between memory use and execution time.

1 Introduction

Side-channel attacks represent a major threat for the security of embedded systems [MOP07]. In particular, they are particularly effective for finding secret keys of block ciphers, such as AES.

The masking countermeasures can provide a strong protection against these attacks [GP99]. The general principle is to split the secret data into several variables, called *shares*, in a way that any strict subset of the shares do not provide information about the secret data. In other words, masking forces an attacker to learn information about all the shares in order to get information about the secret data.

The masking countermeasures require modifying how computations are performed. Indeed, instead of manipulating the secrets directly, one needs to use the shares instead. All the complexity relies on the requirement to avoid revealing any information about the secret during intermediate computations. Depending on the masking scheme used and the computation to perform, this code transformation may be complex, and the resulting overhead can be high.

In this context, finite field multiplication implementation can be critical. Indeed, finite field multiplication is a key operation for various masking types:

- Affine masking [FMPR11] and inner-product masking [BFG15] involve finite field multiplications for masking the secret data.

- Boolean masking can involve many finite field multiplications when masking SBoxes. Multiplications can be used to compute finite field powers, like x^{254} for AES SBox that involves 4 multiplications of masked variables. It can also be used to compute a polynomial that interpolates a given SBox in a finite field [CRV14].

In addition, in the case of boolean masking, each secure multiplication on masked variables involves 4 finite field multiplications *at first order*, i.e. when only the secret is decomposed into 2 shares [RP10]. When increasing the number of shares, the number of finite field multiplication required to performed one multiplication on masked data gets higher and higher.

As such, the number of finite field multiplications required can grow quickly and represent a big part of the overhead of the countermeasure, and there is a strong need for fast constant-time finite field multiplication algorithms.

In [GR17], the authors do an overview of the different methods to implement the finite field multiplication. Among these methods, the exp-log method provides a nice trade-off between execution time and memory consumption. However, its main issue is that it requires particular code to handle the case where one of the operand equals zero, as log is undefined for zero. Until recently, no approach had emerged to implement finite field multiplication with exp-log tables without such particular code for zero.

In this paper, we present an approach that solves this problem. This approach was independently discovered and presented in both [BCH+20] and [CdSGGS22].

We start by recalling the exp-log method in section 2.1. Then, we remind the well-known optimization of the modulo in section 2.2. We explain the technique that avoids any extra code to handle the case where one of the operands equals zero in section 2.3. We compare extensively the exp-log method with this optimization to the other methods for implementing finite field multiplication that were detailed in [GR17], showing the impact on memory (section 3.2), and execution time (section 3.3).

2 Exp-log multiplication and its optimization

2.1 Naïve implementation

The exp-log method is based on the use of a generator element e of the multiplicative group $\mathbb{F}_{2^n}^*$. We have: $\forall x \in \mathbb{F}_{2^n}^*, \exists k, x = e^k$. The number k is called the *log* of x. We can use this generator element to multiply 2 numbers in $\mathbb{F}_{2^n}^*$.

Let `log_t` be a 2^n entry table that contains the logs in base e of all $2^n - 1$ non-zero values (`log_t`$[0]$ is usually set to 0), and `exp_t` a $2^n - 1$ entries table that contains e raised to the power of all values in $[\![0, 2^n - 2]\!]$. Then the multiplication of two **non-zero** elements a and b is given by[1]:

$$a.b = \texttt{exp_t}[(\texttt{log_t}[a] + \texttt{log_t}[b])\%(2^n - 1)]$$

Some code is needed to handle the case where a or b equals zero. In [GR17], the authors propose a sequence of 4 ARM instructions to create a bitmask that equals zero if and only if one of the operands equals zero (asr_{32} designates

[1] % designates the modulo and & designates the `AND` operator

the arithmetic shift right by 32bits, that propagates the sign bit): $bitmask = asr_{32}(-(asr_{32}(-a)\&b))$

The multiplication of 2 numbers of $\mathbb{F}_{2^n}$ is given by:

$$\begin{aligned} a.b =& asr_{32}(-(asr_{32}(-a)\&b)) \\ & \& \texttt{exp_t}[\ (\texttt{log_t}[a] + \texttt{log_t}[b])\ \%\ (2^n - 1)\ ] \end{aligned}$$

2.2 Removal of the modulo

A first optimization to make the implementation faster consists in removing the modulo [GR17]. To achieve this goal, a wider `exp_t` table with $2 * (2^n - 1)$ entries is used. It is defined as follows:

$$\forall x \leq 2^{n+1} - 2, \texttt{exp_t}[x] = e^x$$

The multiplication of 2 numbers of $\mathbb{F}_{2^n}$ with this optimization is given by:

$$\begin{aligned} a.b = & asr_{32}(-(asr_{32}(-a)\&b)) \\ & \& \texttt{exp_t}[\texttt{log_t}[a] + \texttt{log_t}[b]] \end{aligned}$$

2.3 Removal of the code that handles the zero case

We and Cardoso dos Santos et al. proposed an optimization to remove the code that handles the case where one operand equals zero, at the price of a wider `exp_t` table [BCH+20, CdSGGS22]. Note that this optimization cannot be deployed without the modulo optimization.

The general idea is that the sum of 2 logs is bounded by a maximum value: setting `log_t`[0] higher than this maximum allows to easily detect any multiplication that involves a zero operand.

First, we notice that:

$$\forall a \in \mathbb{F}_{2^n}^*, log(a) \leq 2^n - 2$$

We deduce that:

$$\forall a, b \in \mathbb{F}_{2^n}^*, log(a) + log(b) \leq 2^{n+1} - 4$$

Setting $\texttt{log_t}[0] = 2^{n+1} - 3$, we have:

$$\begin{aligned} \forall a, b \in \mathbb{F}_{2^n}, & ((\texttt{log_t}[a] + \texttt{log_t}[b] \geq 2^{n+1} - 3) \\ & \Leftrightarrow (a = 0 \text{ or } b = 0)) \end{aligned}$$

With such a value for `log_t`[0], we need the `exp_t` table to be extended as the sum of 2 logs can now be as high as $2 * (2^{n+1} - 3) = 2^{n+2} - 6$. The `exp_t` table becomes a $2^{n+2} - 5$ entries table and is set as follows:

- $\forall x, 0 \leq x \leq 2^{n+1} - 4, \texttt{exp_t}[x] = e^x$
- $\forall x, 2^{n+1} - 3 \leq x \leq 2^{n+2} - 6, \texttt{exp_t}[x] = 0$

The multiplication of 2 numbers of $\mathbb{F}_{2^n}$ is given by:

$$a.b = \texttt{exp_t}[\texttt{log_t}[a] + \texttt{log_t}[b]]$$

Note: `log_t`[0] can be set to other (higher) values if needed, for instance to get a particular hamming weight, at the price of a larger `exp_t` table.

Table 1: Total size of lookup tables (LUT) used in various multiplication algorithms. `ES` is the size necessary to store one element of the field, e.g. byte for integers up to 255.

Method	exp-log naïve	exp-log w/o mod	exp-log w/o zero	karatsuba	half table	full table
LUT size (in `ES`)	$2^{n+1}-1$	$3*2^n-2$	$c*2^n-5$ $c=5$ or $c=6$	$3*2^n$	$2^{\frac{3n}{2}+1}$	2^{2n}

3 Comparison to alternative table-based implementations

3.1 Alternative approaches considered

In order to evaluate the interest of the optimization, we compare in the next sections its memory impact and number of cycles with other table-based finite field multiplication implementation from [GR17].

We consider the following methods: (1) exp-log tables with optimizations previously presented, (2) karatsuba, (3) half-table, (4) full table. In this section we briefly describe the general ideas of each method. Full algorithms descriptions can be found in [GR17].

The karatsuba method involves 3 lookup tables. Its main principle is to split the multiplication into computations involving smaller numbers represented on half as many bits. For instance, to multiply to bytes, this methods uses 4-bits numbers. Each table access combines 2 of those numbers.

The half-table method involves 2 lookup tables. Similarly to karatsuba, it splits the multiplication into computations involving smaller numbers represented on half as many bits. Yet, instead of combining 2 numbers, each table access depends on 3 numbers.

The full-table method consists in fully tabulating the multiplication, for all possible values of both operands. The multiplication is then done in one table access.

3.2 Memory impact

In this section we compare the size of the lookup tables of considered methods, for a finite field of 2^n elements.

Table 1 presents the total memory size taken by the lookup table of the various methods. The sizes are reported as multiples of the size taken by a single element, i.e. as a number of bytes for $n \leq 8$, as halfwords for $8 < n \leq 16$, and as words for $16 < n \leq 32$. Our method is reported as "exp-log w/o zero" in the table. It has a variable overhead depending on n: $c = 5$ except if n is equal to 8 or 16 or 32. In these cases, $c = 6$. Indeed, for these cases, we need to change the size of the elements in the `log_t` table due to the larger value of `log_t`[0].

Karatsuba and all exp-log methods including ours feature a table size in $O(2^n)$. This is not the case for half-table and full-table: with these methods the table size evolve in $O(2^{\frac{3n}{2}})$ and $O(2^{2n})$ respectively.

Table 2: Total size of lookup tables used in various multiplication algorithms, in bytes, for various n

	exp-log naïve	exp-log w/o mod	exp-log w/o zero	kara-tsuba	half table	full table
n=1	3 B	4 B	5 B	6 B	6 B	4 B
n=2	7 B	10 B	15 B	12 B	16 B	16 B
n=3	15 B	22 B	35 B	24 B	45 B	64 B
n=4	31 B	46 B	75 B	48 B	128 B	256 B
n=5	63 B	94 B	155 B	96 B	362 B	1 KB
n=6	127 B	190 B	315 B	192 B	1 KB	4 KB
n=7	255 B	382 B	635 B	384 B	3 KB	16 KB
n=8	511 B	766 B	1.5 KB	768 B	8 KB	64 KB
n=9	2 KB	3 KB	5 KB	3 KB	45 KB	512 KB
n=10	4 KB	6 KB	10 KB	6 KB	128 KB	2 MB
n=11	8 KB	12 KB	20 KB	12 KB	362 KB	8 MB
n=12	16 KB	24 KB	40 KB	24 KB	1 MB	32 MB
n=13	32 KB	48 KB	80 KB	48 KB	3 MB	128 MB
n=14	64 KB	96 KB	160 KB	96 KB	8 MB	512 MB
n=15	128 KB	192 KB	320 KB	192 KB	23 MB	2 GB
n=16	256 KB	384 KB	768 KB	384 KB	64 MB	8 GB

To make the comparison easier, the actual total size of lookup tables in bytes are reported in Table 2 for various n. While our optimization make the lookup table size clearly bigger than exp-log without this optimization and karatsuba method, the memory impact remains way lower than the one of half-tables.

3.3 Instruction count and estimated number of clock cycles

We compare the number of ARM assembly instructions necessary for the different methods, following the implementations provided by [GR17]. Figure 1 shows the assembly of exp-log method with our optimization. In order to save a load instruction, the exp table is stored right after the load table in memory. The address to the exp table is thus retrieved from the address of the log table, with an addition. Note that the offset used in this addition has to be adjusted depending on the field size, as logs may be stored as halfwords or words instead of bytes.

The measured numbers are presented in Table 3. For each implementation, we indicate the number of arithmetic instructions (ALU), and the number of loads (LDR). We estimate the clock cycles of each implementation by counting 1 clock cycle per ALU instruction and 3 clock cycles per LDR instruction, similarly as in [GR17]. We also estimate the number of clock cycles without initialization, i.e. the number of clock cycles the multiplication would take if memory addresses of the various tables were already in registers. This number makes sense when several multiplications are done in a row. Such case happens for instance when a multiplication is done on masked data, requiring several finite field multiplications under the hood.

Our method has an estimated number of clock cycles per multiplication that

```
log_exp_opt $a, $b, $res,
              $tmp, $ptrlog, $ptrexp

;; init phase
ldr $ptrlog, =log_t
add $ptrexp, $ptrlog, #(2^n)

;; log(a) + log(b)
ldr $tmp, [$ptrlog, $a]
ldr $res, [$ptrlog, $b]
add $tmp, $res

;; res <- exp(tmp)
ldrb $res, [$ptrexp, $tmp]
```

Figure 1: Implementation of exp-log multiplication with our optimization

Table 3: Number of ARM assembly instruction for each method. Loads and ALU operations are separated as their performance impact is different. An estimated number of clock cycles is computed, considering 1 clock cycle per ALU instruction and 3 clock cycles per load instruction.

		exp-log naïve	exp-log w/o mod	exp-log w/o zero	kara-tsuba	half table	full table
# LDR	init only	1	1	1	1	1	1
	w/o init	3	3	3	3	2	1
	total	4	4	4	4	3	2
# ALU	init only	1	1	1	2	1	1
	w/o init	7	5	1	7	3	1
	total	8	6	2	9	4	1
cycles	full	20	18	14	21	13	7
	w/o init	16	14	10	16	9	4

is largely improved compared to previous exp-log methods and to the karatsuba method. It is also very close to the estimated number of clock cycles of the half-table method, both with and without initialization: in both cases, our method is only one clock cycle slower than the half-table implementation. As such, our method provides a good trade-off between impact on memory and on execution time: while its execution time is almost on par with half-table, its memory impact is much lower.

4 Conclusion

This short paper presents an optimization of the finite field multiplication with exp-log tables that handles the case where an operand equals zero by expanding the exp table and by setting a particular value for 0 in the log table. Compared with other table-based methods to implement the finite field multiplication, this

optimization makes the exp-log implementation very competitive. It becomes almost as fast as the half-table implementation while having a way lower impact on memory, whatever the size of the finite field.

5 Acknowledgements

This work was funded as part of the SARMENTI project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825325, and by the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-10-AIRT-05).

References

[BCH+20] Nicolas Belleville, Damien Couroussé, Karine Heydemann, Quentin Meunier, and Inès Ben El Ouahma. Maskara: Compilation of a masking countermeasure with optimized polynomial interpolation. *IEEE TCAD*, 39(11):3774–3786, 2020.

[BFG15] Josep Balasch, Sebastian Faust, and Benedikt Gierlichs. Inner product masking revisited. In *EUROCRYPT*, pages 486–510. Springer Berlin Heidelberg, 2015.

[CdSGGS22] Luan Cardoso dos Santos, François Gérard, Johann Großschädl, and Lorenzo Spignoli. Rivain-Prouff on Steroids: Faster and Stronger Masking of the AES. In *CARDIS*, pages 123–145, Cham, 2022. Springer International Publishing.

[CRV14] Jean-Sébastien Coron, Arnab Roy, and Srinivas Vivek. Fast evaluation of polynomials over binary finite fields and application to side-channel countermeasures. In *CHES*, pages 170–187. Springer, 2014.

[FMPR11] Guillaume Fumaroli, Ange Martinelli, Emmanuel Prouff, and Matthieu Rivain. Affine Masking against Higher-Order Side Channel Analysis. In *Selected Areas in Cryptography*, pages 262–280, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[GP99] Louis Goubin and Jacques Patarin. DES and Differential Power Analysis (The "Duplication" Method). In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, pages 158–172, London, UK, UK, 1999. Springer-Verlag.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In *EUROCRYPT 2017*, pages 567–597, Cham, 2017. Springer International Publishing.

[MOP07] S. Mangard, E. Oswald, and T. Popp. *Power Analysis attacks: Revealing the secrets of smart cards*. Power Analysis Attacks: Revealing the Secrets of Smart Cards. 2007. DOI: 10.1007/978-0-387-38162-6.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In *CHES*, volume 6225, pages 413–427. Springer Berlin Heidelberg, 2010.