Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON

Junwei Wang[1,2], Praveen Kumar Vadnala[2], Johann Großschädl[2], and Qiuliang Xu[1]⋆

[1] School of Computer Science and Technology,
Shandong University, Jinan 250100, Shandong, China
`wakemecn@gmail.com, xql@sdu.edu.cn`

[2] Laboratory of Algorithmics, Cryptology and Security,
University of Luxembourg, Luxembourg
`{praveen.vadnala,johann.groszschaedl}@uni.lu`

Abstract. Real-world software implementations of cryptographic algorithms need to be able to resist various kinds of side-channel attacks, in particular Differential Power Analysis (DPA). Masking is a widely-used countermeasure to protect block ciphers like the Advanced Encryption Standard (AES) against DPA attacks. The basic principle is to split all sensitive intermediate variables manipulated by the algorithm into two shares and process these shares separately. However, this approach still succumbs to higher-order DPA attacks, which exploit the joint leakage of a number of intermediate variables. A viable solution is to generalize masking such that at least $d + 1$ shares are used to protect against d-th order attacks. Unfortunately, all current higher-order masking schemes introduce a significant computational overhead compared to unmasked implementations. To facilitate the deployment of higher-order masking for the AES in practice, we developed a vector implementation of Coron et al's masking scheme (FSE 2012) for ARM NEON processors. After a comprehensive complexity analysis, we found that Coron et al's scheme with n shares for each sensitive variable needs $\mathcal{O}(n^2)$ multiplications in the field $\mathrm{GF}(2^8)$ and $\mathcal{O}(n^2)$ random-number generations. Both of these performance-critical operations are executed with only 15 instructions in our software, which is possible thanks to the rich functionality of the NEON instruction set. Our experimental results clearly indicate that the performance penalty caused by the integration of higher-order masking is significantly lower than in generally assumed and reported in previous papers. For example, our second-order DPA-protected AES (with three shares for each sensitive variable) is merely eight times slower than an unmasked baseline implementation that resists cache-timing attacks.

1 Introduction

Differential Power Analysis (DPA) [3] is a cryptanalytic technique that exploits variations in the power consumption of a cryptographic device (e.g. smart card)

⋆ Corresponding author

to obtain the secret key. DPA attacks first appeared in the literature in the late 1990s [15] and have since then received much attention from the cryptographic research community. Basically, a DPA attack consists of two phases, namely an acquisition phase and an analysis phase. In the former phase, the attacker has to acquire a set of power consumption traces of the target device while it executes a cryptographic algorithm with different inputs (i.e. different plaintexts or ciphertexts). Then, in the analysis phase, he applies sophisticated statistical techniques to determine the correlation between the measured power traces and certain intermediate values of the cryptographic algorithm based on the known inputs and a predicted (i.e. guessed) part of the secret key. More precisely, the attacker uses the computed intermediate values to partition the power traces into several categories and determines the correlation (e.g. the difference in the averages of the categories) between the measured traces and the intermediate values. Normally, the highest correlation is obtained with a partitioning where the guessed key is the actual secret key processed by the device [16].

Masking is a widely-used countermeasure to protect block ciphers, such as the AES, against DPA attacks by randomizing all sensitive intermediate values. The basic idea is to split the intermediate values into multiple shares, and then process all operations on each share separately throughout the execution of the algorithm. At the very end of the algorithm, the shares need to be recombined to get the correct result. For example, a sensitive value x can be split into two shares x_1 and x_2, where x_1 is a freshly generated random value and x_2 is computed as $x_2 = x \otimes x_1$, based on the underlying operation $\otimes$[3]. Since each masked value is pairwise independent from the sensitive value and the mask, the power consumption does not depend on the sensitive value any more, i.e., information leaked from the mask alone or the masked value alone cannot be used to reveal the secret.

DPA attacks involving a single independent variable are also known as first-order DPA attacks. High-order DPA attacks exploit the joint leakage of several variables manipulated in the device. Here, the order denotes the number of intermediate variables exposed to the attacker. First-order DPA secure implementation with masking countermeasures is still vulnerable to second-order DPA attacks [5, 17, 21], which exploit combined leakage from two intermediate variables with increased effort in the number of required samples and computation complexity. For example, if two points in the power traces are correlated to the intermediate values x and y, where the secret $z = x \oplus y$, then the combination of leakage information of the two intermediate values can be used to predict the secret. In general, an n-th order masking scheme can be defeated by an $(n+1)$-th order DPA attack [4].

Theoretically, no countermeasure can fully protect a cryptosystem against all high-order DPA attacks. Nevertheless, the effort to successfully mount a high-order DPA attack grows exponentially as the order increases. Hence, taking into account the attacker's ability, it is sufficient in practice the implementation to be resistant against DPA attacks up to a certain order. Generally, in a d-th

[3] In this paper, we only focus on Boolean masking and hence $\otimes$ is replaced with $\oplus$.

order masking scheme, every sensitive value is split into $n = d + 1$ shares such that $x = x_1 \oplus \cdots \oplus x_n$, where $x_1, \cdots, x_{n-1}$ are independently generated random variables called masks, and $x_n = x \oplus x_1 \oplus \cdots \oplus x_{n-1}$ is the masked data. When a d-th order masking scheme is executed, each share x_i is processed separately to ensure that the combination of any d shares is independent of the sensitive variable, i.e., the combination of d leakages does not leak any information about the sensitive variable. As a consequence, a d-th order DPA attack exploiting up to d joint leakages can not reveal the secret key anymore.

The AES consists of several rounds, each performing a few linear transformations and a non-linear transformation. The linear part can be trivially masked by applying the transformations on each share separately since $f(x) = f(x_1) \oplus \cdots \oplus f(x_n)$, where $f(\cdot)$ denotes a linear function and $x = x_1 \oplus \cdots \oplus x_n$. However, the real difficulty is the secure masking of the non-linear transformation (also called substitution, or S-box) $S(\cdot)$. At CHES 2010, Rivain and Prouff [18] presented a generic masking scheme for the AES to make it secure against d-th order attacks. However, Coron *et al.* later showed that the scheme is not resistant to d-th order attacks as claimed and provided a fix [7]. A comprehensive treatment of other higher-order masking schemes proposed in the literature can be found in [11, 10]. The main problem with all existing higher-order masking schemes is that they are extremely slow, typically hundreds of magnitude slower compared to an unmasked implementation. In this paper, we try to bridge this gap through a combination of algorithmic improvements and highly-optimized implementation. The algorithmic improvements are obtained by using Barret reduction for the field multiplication and composite field arithmetic to compute the inverse of a field element. The improvement in implementation is obtained by using vectorization, thanks to the richness of ARM NEON instruction set .

We first describe a baseline AES implementation resistant against cache-timing attacks (but without DPA countermeasures) using NEON instructions. The S-box in our baseline implementation is computed over composite field. Compared with Gladman's table-based implementation for 32-bit processors [9], our baseline implementation is only marginally slower, but resists cache-timing attacks. Hence, it is a reasonable starting point for comparisons in practice. We then develop the first vector implementation of high-order DPA countermeasure for AES using the ARM NEON instruction set. Our implementation is based on the Coron *et al.*'s fix for the RP countermeasure. Finally we apply composite field arithmetic to further improve the efficiency. Based on a thorough complexity analysis, we found that the most performance-critical parts are the field multiplications and pseudorandom number generations, both of which only take 15 instructions in our vector implementation.

We did a careful execution time evaluation of various different AES implementations using a cycle-accurate instruction set simulator. We found that the performance penalty due to the integration of high-order DPA countermeasures in our implementation is much lower than in previous works. For example, our results show that an optimized NEON assembly implementation of the AES with integrated second-order countermeasures (using three shares for each sensitive variable) is only eight times slower than a baseline implementation without DPA countermeasures, which means our implementation is efficient enough to be used in practice.

2 Previous Work

In this section, we recall the existing algorithms for secure higher order masking of AES. We first review the higher-order masking scheme proposed by Prouff and Rivain [18]. Then we describe the flaw in the original algorithm and its fix due to Coron *et al.* [7].

2.1 Provably Secure Higher-order masking of AES

The first solution to secure AES against higher-order attacks was proposed by Prouff and Rivain at CHES 2010 [18]. Their solution was based on the higher-order masking scheme proposed by Ishai, Sahai and Wagner (ISW) to protect any circuit against t-limited adversary, who can tap any t wires in the circuit at a given time [13]. The main idea of ISW scheme is to represent the circuit which performs the cryptographic operations as a combination of Boolean gates AND and NOT (which is possible since NAND is a universal gate), and protect these gates independently. Securing NOT gate is easy since $NOT(x_1 \oplus \cdots \oplus x_n) = NOT(x_1) \oplus \cdots \oplus x_n$. To protect AND gate, they proposed an elegant solution where the inputs are elements in $GF(2)$. However, Prouff and Rivain later shown that this method can actually be extended to secure multiplication over any field of characteristic 2: $GF(2^n)$ (as a result also to AES field: $GF(2^8)$). We recall their solution in Algorithm 1. They also reduced the number of shares required for d-th order secure masking scheme to $d + 1$ from $2d + 1$ required for ISW method.

Algorithm 1 SecMult - Masked multiplication over $GF(2^8)$ with n shares [6]

```
Input: shares $x_i$ satisfying $x_1 \oplus \cdots \oplus x_n = x$, shares $y_i$ satisfying $y_1 \oplus \cdots \oplus y_n = y$
Output: shares $z_i$, satisfying $z_1 \oplus \cdots \oplus z_n = xy$
 1: for $i = 1$ to $n$ do
 2:     $z_i \leftarrow x_i y_i$
 3: end for
 4: for $i = 1$ to $n$ do
 5:     for $j = i + 1$ to $n$ do
 6:         $r_0 \leftarrow \mathbb{F}_{2^8}$
 7:         $r_1 \leftarrow (r_0 \oplus x_i y_j) \oplus x_j y_i$
 8:         $z_i \leftarrow z_i \oplus r_0$
 9:         $z_j \leftarrow z_j \oplus r_1$
10:     end for
11: end for
12: return $z_1, \cdots, z_n$
```

Masking linear function $f(.)$ is easy since

$$f(x) = f(x_1) \oplus f(x_2) \oplus \cdots \oplus f(x_n) \tag{1}$$

The only non-linear function in AES S-box is the inversion over the finite field $GF(2^8)$. Since the inversion x^{-1} of an element x in the finite field $GF(2^8)$ equals x^{254}, the authors perform the secure computation of inversion through secure exponentiation, which comprises several secure field multiplications and squarings. To securely mask the field multiplications in $\mathbb{F}_2^8$, the authors extended the techniques for masking a logical AND (that is, field multiplication in $\mathbb{F}_2$) proposed in [13], which can be applied to securely mask a multiplication over any field of characteristic 2. To protect the AES power function $x \to x^{254}$, Prouff and Rivain gave a solution which is recalled in Appendix A. Later Kim *et al.,* improved the efficiency using composite field arithmetic [14].

2.2 Higher-order Side-channel Security and Mask Refreshing

Algorithm 5 uses **RefreshMasks** procedure before performing **SecMult** on dependent inputs. [4] The **RefreshMasks** procedure basically modifies the shares using freshly generated randoms. Let $(r_i)_{1\leq i\leq d}$ be the new random numbers. Then a call to **RefreshMasks** $((x_i)_{1\leq i\leq d+1})$ performs the following operation:

$$\begin{aligned} x_0 &= x_0 \oplus \bigoplus_{1\leq i\leq d} r_i \\ (x_i)_{1\leq i\leq d} &= x_i \oplus r_i \end{aligned}$$

Though the procedures **RefreshMasks** and **SecMult** are secure against d-th order attacks independently, Coron *et al.* pointed out that a flaw arises when they are integrated to compute $x \cdot g(x)$, where $g(\cdot)$ is a linear function [7]. The authors showed that a joint leakage of $\lfloor n/2 \rfloor + 1$ intermediate variables can be exploited due to the involvement of the **RefreshMasks** procedure in certain cases. In other words, the claim that the RP countermeasure is secure against d-th order attacks is not valid any more, where $d = n-1$. In order to prevent this flaw, a new d-th order masked multiplication of form $x \cdot g(x)$ was put forward in the same paper. In detail, suppose

$$f(x, y) = (x \cdot g(y)) \oplus (g(x) \cdot y)$$

where $x, y \in GF(2^8)$ and $g(\cdot)$ is a linear function over $GF(2^8)$, then

$$f(x, y) = f(x, r) \oplus f(x, y \oplus r)$$

[4] The function **SecMult** is only secure when the inputs are d-independent of each other. Namely, every $2d$-tuple containing d elements from the input x $((x_i)_{1\leq i\leq d+1})$ and d elements from the input y $((y_i)_{1\leq i\leq d+1})$ should be uniformly distributed and independent of x and y.

due to the bilinearity of $f(\cdot)$ [7]. Hence, the flaw is fixed by recalculating the r_1 in Line 7 of Algorithm 1 as follow,

$$\begin{aligned} r_1 &= r_0 \oplus f(x_i, x_j) \\ &= (r_0 \oplus f(x_i, r'_0)) \oplus f(x_i, x_j \oplus r'_0) \end{aligned}$$

where r'_0 is a freshly generated random value in $GF(2^8)$. The linear memory complexity version of secure computing $h = x \cdot g(x)$ is recalled in Algorithm 2. For each pair of shares x_i, x_j, this algorithm requires generation of an extra

Algorithm 2 SecH - masked multiplication $h(x) = x \cdot g(x)$ over $GF(2^8)$ with n shares, where $g(\cdot)$ is a linear function [7]

```
Input: shares $x_i$ satisfying $x_1 \oplus \cdots \oplus x_n = x$, and $g(x_i)$ for each $x_i$
Output: shares $y_i$, satisfying $y_1 \oplus \cdots \oplus y_n = x \cdot g(x)$
 1: for $i = 1$ to $n$ do
 2:     $y_i \leftarrow x_i \cdot g(x_i)$
 3: end for
 4: for $i = 1$ to $n$ do
 5:     for $j = i + 1$ to $n$ do
 6:         $r_0 \leftarrow GF(2^8)$
 7:         $r'_0 \leftarrow GF(2^8)$
 8:         $r_1 \leftarrow r_0 \oplus (x_i \cdot g(r'_0)) \oplus (r'_0 \cdot g(x_i)) \oplus (x_i \cdot g(x_j \oplus r'_0)) \oplus ((x_j \oplus r'_0) \cdot g(x_i))$
 9:         $y_i \leftarrow y_i \oplus r_0$
10:         $y_j \leftarrow y_j \oplus r_1$
11:     end for
12: end for
13: return $y_1, \cdots, y_n$
```

random number to split the computation of $f(x_i, x_j)$. Besides, two more field squarings and two more multiplications are required for Line 8. The authors also showed that by using look-up tables for the function $h(x) = x \cdot g(x)$, one could improve the efficiency. However, the implementations are still several magnitudes (in the order of hundreds) slower than unmasked implementations and are not suitable for most of the applications. In this paper, we take first steps towards bringing these schemes closer to practice.

3 Implementation

In this section, we describe our implementation of higher-order secure AES from [7] on ARM NEON. We first give our efficient solution to implement the field multiplication in $GF(2^8)$, which is the costliest operation in securing AES according to our comprehensive analysis, and then describe a method to secure the full cipher.

3.1 Vector Implementation of Field Multiplication in $GF(2^8)$

Barrett Reduction over Integers In order to optimize the modular reduction

$$r = a \texttt{ mod } n, \tag{2}$$

where a, n are integers and $a < n^2$, Barrett proposed an efficient algorithm at CRYPTO 1986 [1]. Barrett reduction is designed to replace the trial division with multiplications, which is expected to be much faster.

The general idea of Barrett reduction is based on the following equation

$$a \texttt{ mod } n = a - \left\lfloor \frac{a}{n} \right\rfloor n. \tag{3}$$

One can precompute $m = 1/n$, in which case the modular reduction operation is transformed into two multiplications and one subtraction. However, since the quotient $m = 1/n$ can only be represented as a floating point number and Barrett's algorithm is supposed to work with integers, it adopts a trick to avoid calculating on floating numbers. Suppose k is the minimal integer such that $2^k > n$, which means we can precompute $m = \lfloor 2^{2k}/n \rfloor$. Let $q = \lfloor ma/4^k \rfloor$, $r = a - qn$ then $\lfloor a/n \rfloor - 1 < q \leq \lfloor a/n \rfloor$ and

$$a \texttt{ mod } n = \begin{cases} r & \text{if } r < n, \\ r - n & \text{otherwise.} \end{cases} \tag{4}$$

The entire algorithm requires two multiplications, one shift operation[5] and at most two subtractions.

Modular Reduction in $\mathbb{F}_q[x]$ In order to perform modular reduction over $\mathbb{F}_q[x]$, Dhem generalized Barrett's modular reduction over integers to work with polynomials [8]. Theorem 1 indicates that Barrett modular reduction can be adapted to extension fields with polynomial presentation.

Theorem 1 (Quotient Evaluation in $\mathbb{F}_q[x]$, adpated from [8]). *Suppose $U(x)$, $N(x)$, $Z(x)$ and $Q(x)$ are polynomials over $\mathbb{F}_q$, and $U(x) = Q(x)N(x) + Z(x)$ (i.e., $Z(x) = U(x) \texttt{ mod } N(x)$), then*

$$Q(x) = \left\lfloor \frac{U(x)}{N(x)} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{U(x)}{x^p} \right\rfloor \left\lfloor \frac{x^{p+\beta}}{N(x)} \right\rfloor}{x^\beta} \right\rfloor = \left\lfloor \frac{T(x)R(x)}{x^\beta} \right\rfloor, \tag{5}$$

where $p = deg(N(x))$[6], $\beta \geq deg(U(x)/x^p)$ and $\lfloor A(x)/B(x) \rfloor$ stands for the quotient of polynomial division $A(x)/B(x)$, ignoring the reminder.

According to Theorem 1, we can perform the modular reduction operation $U(x) = Z(x) \texttt{ mod } N(x)$ over $\mathbb{F}_q[x]$ in three steps as follows.

[5] The division operation in which 4^k is the divisor can be calculated as right-shift of $2k$ bits.

[6] $deg(A(x))$ stands for the degree of polynomial $A(x)$.

Step 1. Evaluate the quotient $Q(x) = \lfloor U(x)/N(x) \rfloor$ according to Theorem 1.
Step 2. Calculate the product $V(x) = Q(x)N(x)$.
Step 3. Obtain the reminder $Z(x) = U(x) - V(x)$.

In most applications, $N(x)$ is fixed, e.g., $N(x) = x^8+x^4+x^3+x+1$ in the case of AES. Therefore, we can accelerate the computation of $Q(x)$ by pre-computing $R(x) = \lfloor x^{p+\beta}/N(x) \rfloor$. Although Theorem 1 holds when $\beta \geq deg(U(x)/x^p)$, there is no need to choose $\beta > deg(U(x)/x^p)$, because the bigger β, the bigger $R(x)$ and the more computation is required. In general, we can choose $\beta = \alpha - p$, where $deg(U(x))$ is bounded by some constant value α. Thus, the evaluation of $Q(x)$ is simplified to one multiplication and two shift operations (i.e., division by x^p and x^β, similar to the case of integers [5]). Overall, a Barrett modular reduction for polynomials over $\mathbb{F}_q[x]$ consists of two multiplications, two shift operations and one subtraction.

Field Multiplication in $\mathbb{F}_{2^8}$ (i.e., $GF(2^8)$) A complete field multiplication in $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/q(x)$, where $q(x) = x^8 + x^4 + x^3 + x + 1$, consists of two steps: multiplying two polynomials $a(x)$ and $b(x)$ of degree ≤ 7 and modular reduction of the product $p(x)$ with respect to $q(x)$. Since we have `vmull.p8` instruction in NEON, a polynomial multiplication can be easily carried out in parallel. Hence, the only operation we have to pay attention to is the modular reduction.

Polynomial operations over $\mathbb{F}_2[x]$ have some special characteristics (given below), which can be used to speed up the modular reduction operation. [7]

1. A polynomial of degree $m-1$ can be represented by an array of m bits.
2. The subtraction of two polynomials is same as addition.
3. The product of a polynomial of degree $m-1$ and a polynomial of degree $n-1$ is a polynomial of degree $m+n-2$, which can be represented by $(m+n-1)$ bits. In the case of $GF(2^8)$, we have $m, n \leq 7$, and therefore $m+n-2 \leq 12$. However, for two integers consisting of m and n bits, the product has a length of $(m+n)$ bits.
4. The addition of two polynomials of degree m is a polynomial of degree m. In case of two integers of m-bit length, the addition may have a length of $(m+1)$ bits.
5. The reminder of division of two polynomials is one degree smaller than the divisor (modulus). In the case of $GF(2^8)$, the degree of the irreducible polynomial $q(x)$ is 8. However, in the case of integers, the reminder might have the same length as the divisor (in binary representation).

Using the observations above, we designed Algorithm 3, which realizes field multiplications in $GF(2^8)$.

In the pre-computations stage, we know that the degree of the product of two polynomials of degree $\leq p-1$ cannot exceed $\alpha = 2*(p-1)$. Here, $p = deg(N(x)) = 8$ and hence $\beta \geq \alpha - p = 6$. In order to optimize the computation, we can choose an $R(x)$ that can be stored in a single byte, which means $R(x)$ should have a degree of ≤ 7. Since $R(x) = \lfloor x^{p+\beta}/N(x) \rfloor$, we can for example pick $\beta = 6$ and $R(x) = x^6 + x^2 + x$. Alternatively, we can pick $\beta = 7$ and

[7] Some of the characteristics are also valid in $\mathbb{F}_q[x]$, where $q > 2$.

Algorithm 3 Field multiplication in $GF(2^8)$

Input: Polynomials $A(x)$ and $B(x)$ in $GF(2^8)$
Output: Polynomial $Z(x) = A(x) \cdot B(x) \bmod N(x)$, where $N(x) = x^8 + x^4 + x^3 + x + 1$.

```
Pre-computation:
1: $p \leftarrow deg(N(x))$                              ▷ $p = 8$
2: $\alpha \leftarrow 2 * (p - 1)$                       ▷ $\alpha = 14$
3: $\beta \geq \alpha - p$                               ▷ $\beta \geq 6$
4: $R(x) \leftarrow \lfloor \frac{x^{p+\beta}}{N(x)} \rfloor$   ▷ $R(x) = x^6 + x^2 + x$ if $\beta = 6$
Multiplication with Barrett modular reduction(Theorem 1):
1: $U(x) \leftarrow A(x) \cdot B(x)$                     ▷ $deg(U(x)) \leq 14$
2: $T(x) \leftarrow \lfloor \frac{U(x)}{x^p} \rfloor$    ▷ $deg(T(x)) \leq 6$
3: $S(x) \leftarrow T(x) \cdot R(x)$                     ▷ $deg(S(x)) \leq \beta + 6$
4: $Q(x) \leftarrow \lfloor \frac{S(x)}{x^\beta} \rfloor$ ▷ $deg(Q(x)) \leq 6$
5: $V(x) \leftarrow Q(x) \cdot N(x)$                     ▷ $deg(V(x)) \leq 14$
6: $Z(x) \leftarrow U(x) + V(x)$
```

$R(x) = x^7 + x^3 + x^2 + 1$. However, for $\beta \geq 8$, $R(x)$ has a degree ≥ 8, which is not desirable.

In total, one field multiplication requires three polynomial multiplications, two shift operations and one addition. Moreover, the execution sequence of this technique for field multiplication is independent of the processed operands and, hence, it is resistant against timing attacks.

Vector Implementation of Field Multiplication in $GF(2^8)$ Since the AES state consists of 16 bytes, we aim at vectorizing the transformations on all the 16 bytes. In order to perform 16 field multiplications in parallel, we define a function named `fmult` as follows:

```
uint8x16_t fmult(uint8x16_t a, uint8x16_t b);
```

The `fmult` function takes two arguments of type `uint8x16_t`[8] (i.e. a vector of 16 bytes) and returns a vector of 16 bytes. The most and least significant eight bytes of the vector $U(x) = A(x) \cdot B(x)$ can be calculated in parallel using `vmull.p8` instruction. To compute $T(x) = \lfloor \frac{U(x)}{x^8} \rfloor$, we right shift every element in $U(x)$ by 8 bits with help of the `vshrn.i16` instruction. The vectors $S(x) = T(x) \cdot R(x)$ and $Q(x) = \lfloor \frac{S(x)}{x^6} \rfloor$ can similarly be calculated using `vmull.p8` and `vshrn.i16` respectively. Since the most significant byte of each element in $V(x)$ and $U(x)$ are the same so that they cancel out each other in the last step (namely, $V(x)+U(x)$), we only need to calculate the least significant byte of each element of $V(x)$. Since $V(x) = Q(x) \cdot N(x)$, where $N(x) = x^8 + x^4 + x^3 + x + 1$ is 100011011 in binary

[8] In NEON jargon, `uint8x16_t` is a quadword vector of sixteen unsigned integers of 8-bit length

presentation (0x11B in hexadecimal format), we have

$$V(x) \bmod x^8 = (Q(x) \bmod x^8) \cdot (N(x) \bmod x^8).$$

Since $deg(Q(x)) \leq 7$ and $N(x) \bmod x^8 = x^4 + x^3 + x + 1$, the least significant byte of $V(x)$, i.e., $V(x) \bmod x^8$, is calculated in the following way,

$$V(x) \bmod x^8 = (Q(x) \bmod x^8) \cdot (x^4 + x^3 + x + 1).$$

Finally, the `veor` instruction conducts the XOR operation (addition) of the least significant byte of each element in $U(x)$ and $V(x)$. Consequently, only 15 instructions are used in our vector implementation of the field multiplication in $GF(2^8)$, which is less than one instruction per byte.

3.2 Vector Implementation of Round Operations

We now describe our implementation of all the round operations of AES. [9] It is easy to mask a linear function $f(\cdot)$, since

$$f(x) = f(x_1) \oplus \cdots \oplus f(x_n), \tag{6}$$

where $x = x_1 \oplus \cdots \oplus x_n$. The operations *AddRoundKey*, *ShiftRows* and *MixColumns* are linear and can be implemented in a straightforward way. The non-linear part of the cipher i.e., S-Box consists of an inversion in $GF(2^8)$ and an affine transformation. Masking the affine transformation is similar to masking a linear function. Masking the inversion involves several subroutines: masking the field squaring, masking the field multiplication and masking $h(x) = x \cdot g(x)$, where $g(\cdot)$ is a linear function. We will discuss these subroutines separately below.

AddRoundKey *AddRoundKey* is a linear function, because it is simply an XOR operation. Due to the convenient vector XOR instruction `veor`, we only need one instruction to implement this operation.

ShiftRows *ShiftRows* left-rotates bytes in the n-th row of the state matrix by $(n-1)$ positions.

$$ShiftRows : \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} \mapsto \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{11} & x_{12} & x_{13} & x_{10} \\ x_{22} & x_{23} & x_{20} & x_{21} \\ x_{33} & x_{30} & x_{31} & x_{32} \end{bmatrix} \tag{7}$$

As it only rearranges the order of bytes in an AES state, it is also a linear transformation. We use a lookup table based on the number of shifts required

[9] We do not describe the implementation of key expansion as it can be obtained in a similar way.

for each byte and store it in a static array. We then reorder the state bytes according to the look-up table by using the vector table look-up instruction `vtbl.8`. [10] We require four instructions to implement *ShiftRows* operation, two each for loading the table and reordering the state bytes.

MixColumns *MixColumns* can be performed by left-multiplying each column with a constant byte matrix $\mathcal{M}$ with four rows and four columns, where the multiplication is applied on $GF(2^8)$.

$$MixColumns : \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \mapsto \begin{bmatrix} 2\,3\,1\,1 \\ 1\,2\,3\,1 \\ 1\,1\,2\,3 \\ 3\,1\,1\,2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{8}$$

The multiply by 2 can be realized via a single left-shift and a XOR operation; the multiply by 3 is realized via combination of a multiply by 2 and an XOR operation. The parallel implementation of *MixColumns* resistant against timing attacks (i.e., without conditional branches) can be obtained with only 13 instructions.

Field Squaring. Squaring is a linear operation in $\mathbb{F}_2^n$ and hence can be masked by squaring the shares independently.

Field Multiplication. The vector implementation of the field multiplication can be carried out in a straightforward way using **SecMult** (Algorithm 1) and `fmult` function.

Masking $\boldsymbol{h(x) = x \cdot g(x)}$. Table lookups are a common way to improve the execution time of this operation. To mask $h(x) = x \cdot g(x)$, we can also store a look-up table for $h(x)$. However, we cannot perform vector-parallel look-ups into a table of more than 32 elements with NEON instructions. Hence, in our implementation, we manually compute the values (as in Line 8 of Algorithm 3).

Affine Transformation. Suppose the byte $x = [x_0 x_1 \cdots x_7]$, where $x_0, \cdots, x_7$ are bits, is one of the shares of the multiplicative inverse calculated in the last step. After the affine transformation, x should be modified as follows:

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} = \begin{bmatrix} x_7 \oplus x_6 \oplus x_5 \oplus x_4 \oplus x_3 \\ x_6 \oplus x_5 \oplus x_4 \oplus x_3 \oplus x_2 \\ x_5 \oplus x_4 \oplus x_3 \oplus x_2 \oplus x_1 \\ x_4 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0 \\ x_7 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0 \\ x_7 \oplus x_6 \quad x_2 \oplus x_1 \oplus x_0 \\ x_7 \oplus x_6 \oplus x_5 \oplus x_1 \oplus x_0 \\ x_7 \oplus x_6 \oplus x_5 \oplus x_4 \oplus x_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{9}$$

[10] Note that the table look-up here is not based on the secret key, hence is not vulnerable to cache-timing attacks.

Hence, five steps have to be carried out to implement the affine transformation.

Step 1. Cyclic left shift of x by one bit: $y = [x_7 x_0 x_1 x_2 x_3 x_4 x_5 x_6]$.
Step 2. Cyclic left shift of x by two bits: $z = [x_6 x_7 x_0 x_1 x_2 x_3 x_4 x_5]$.
Step 3. Cyclic left shift of x by three bits: $v = [x_5 x_6 x_7 x_0 x_1 x_2 x_3 x_4]$.
Step 4. Cyclic left shift of x by four bits: $w = [x_4 x_5 x_6 x_7 x_0 x_1 x_2 x_3]$.
Step 5. Finally, $x = x \oplus y \oplus z \oplus v \oplus w \oplus 0\text{x}63$.

4 Improved Implementation of Secure Inversion over Composite Field

The most costly operation in the implementation of the AES S-box is computing the multiplicative inverse over finite field $GF(2^8)$. In order to accelerate the evaluation of inversion operation, several composite field methods were proposed [19, 20]. Kim *et al.,* [14] used this idea to fasten the secure high-order masking of AES S-box proposed by Rivian-Prouff [18]. However, as it also uses **RefreshMaks** procedure, the attack from [7] is also valid here. In this section, we describe a method to overcome the attack.

Composite Field. In a typical composite field method, one first maps an element over $GF(2^8)$ into an element over composite field using an isomorphism function δ. Then, the inversion is computed over the composite field. In the end, the result is transformed back to an element over $GF(2^8)$ by the inverse mapping function δ^{-1}. More precisely, for any element $A = a_h\gamma + a_l$ in composite field $GF((2^4)^2)$, where $a_h, a_l \in GF((2^4)^2)$, the multiplicative inverse of A can be carried out as $A^{-1} = (A^{17})^{-1} \cdot A^{16}$, according to the equation in [12]. Here, A^{16} can be computed by four bitwise XOR operations, since $A^{16} = a_h\gamma + (a_h + a_l)$. The value A^{17} can be obtained by multiplying A and A^{16} over $GF((2^4)^2)$, i.e., $A^{17} = \lambda a_h^2 + (a_h + a_l)a_l$ (since $\gamma^2 + \gamma = \lambda$). Hence, the inversion of $x \in GF(2^8)$ can be completed by performing the following steps:

Step 1. Apply the isomorphism function δ, such that $A = a_h\gamma + a_l = \delta(x) \in GF((2^4)^2)$, where $a_l, a_h \in GF(2^4)$.
Step 2. Compute A^{17} as $d = \lambda a_h^2 + (a_h + a_l)a_l \in GF(2^4)$.
Step 3. Evaluate the inversion of A^{17}, namely, $d' = d^{-1}$.
Step 4. Compute the inversion $A^{-1} = (A^{17})^{-1} \cdot A^{16} = a'_h\lambda + a'_l$ where $a'_h = d'a_h \in GF(2^4)$ and $a'_l = d(a_h + a_l) \in GF(2^4)$.
Step 5. Compute the inversion of x by applying the inverse mapping function δ', i.e., $x^{-1} = \delta'(a'_h\gamma + a'_l)$.

Secure Inversion over Composite Field. Instead of securely raising an element to 254, [14] performs secure inversion by using composite field method, i.e., they securely mask the aforementioned five steps.

As previously mentioned, the linear functions δ and δ' can be masked by simply applying the function on each share separately. The field multiplication in $GF(2^4)$ can be masked in the same way as shown in Algorithm 1. The multiplicative inversion in $GF(2^4)$, i.e., raising the operand to 14, can be implemented as a combination of two linear operations (namely, squaring and raising to power 4) and one secure field multiplication, which is constructed as follows,

$$x \xrightarrow{x^2} x^2 \xrightarrow[\textbf{RefreshMasks}]{x^2 x} x^3 \xrightarrow{(x^3)^4} x^{12} \xrightarrow{x^{12}x^2} x^{14}. \tag{10}$$

All these operations can be directly masked using the techniques proposed in [18]. To implement their solutions on embedded systems, the authors suggest to pre-compute several tables of 16 elements or 256 elements, such as field multiplication table, squaring table and isomorphism function table, which can significantly improve the overall performance. The running times can be further reduced by combining the inverse isomorphism function and affine function.

Our Improved Implementation of Secure Inversion over Composite Field. Due to the involvement of **RefreshMasks** procedure, the secure inversion in [14] is also vulnerable to the attack mentioned in [7]. In order to avoid this attack, we propose a new secure inversion algorithm as shown in Algorithm 4, where **SecH** is a variant of Algorithm 2 over $GF(2^4)$. The security of Algorithm 2 directly follows from the proof given in Section 4 of [7]. To optimize the performance, we store a pre-computed table for the function h in our corrected implementation.

Algorithm 4 SecInv4 - Masked exponentiation to 14 over $\mathbb{F}_{2^4}$ with n shares

```
Input: shares $x_i$ satisfying $x_1 \oplus \cdots \oplus x_n = x$
Output: shares $y_i$ satisfying $y_1 \oplus \cdots \oplus y_n = x^{254}$
1: for $i = 1$ to $n$ do
2:     $w_i \leftarrow x_i^2$                                              ▷ $\bigoplus_i w_i = x^2$
3: end for
4: $(z_1, \cdots, z_n) \leftarrow$ SecH$((x_1, \cdots, x_n), (w_1, \cdots, w_n))$          ▷ $\bigoplus_i z_i = x^3$
5: for $i = 1$ to $n$ do
6:     $z_i \leftarrow z_i^4$                                              ▷ $\bigoplus_i z_i = x^{12}$
7: end for
8: $(y_1, \cdots, y_n) \leftarrow$ SecMult$((z_1, \cdots, z_n), (w_1, \cdots, w_n))$       ▷ $\bigoplus_i y_i = x^{14}$
```

The vector table look-up instruction `vtbl.8` can do a parallelized look-up in a table of at most 32 elements, and hence is not suitable for tables of 256 elements such as the field multiplication table over $GF(2^4)$, the isomorphism and inverse isomorphism tables. For the field multiplication over $GF(2^4)$, we again utilize the Barrett's reduction technique. Compared to Algorithm 3, the algorithm to perform field multiplication over $GF(2^4)$ is simpler. More preciously, we can ignore Step 3 and Step 4 in Algorithm 3, since in the case of $GF(2^4)$, if we choose $\beta = 2$, the polynomial $R(x) = x^\beta = x^2$ and these two steps actually cancel each other and do nothing but set $Q(x) = T(x)$. Besides, all temporary values in the algorithm can be stored in a single byte. Consequently, only 6 instructions are used in our vector implementation of the field multiplication in $GF(2^4)$, which is much faster than table look-up. We present our algorithm to perform field multiplication over $GF(2^4)$ in Appendix B.

5 Implementation Results

5.1 Baseline Implementation

For performance comparison, we need a baseline implementation that is resistant against timing attacks, i.e., we need an implementation that does not use look-up tables. Hence, we developed a baseline implementation using the ARM NEON instruction set from scratch, which performs the inversion in $GF(2^8)$ by using composite field method. In fact, the baseline implementation is exactly the implementation that we mentioned in Section 4 where only one share (i.e., without any freshly generated random masks) is involved. To achieve better performance, we optimized the baseline implementation with pure NEON assembly language and unrolled all the loops, i.e., we eliminated all avoidable loss of efficiency.

Usually, Gladman's AES implementation [9] is used as a starting point for comparison. However, his implementation [9] uses look-up tables, and is vulnerable to cache-timing attacks. Hence, it is not suitable as baseline implementation for comparison, even though it achieves very good execution time. Nevertheless, we do a comparison between Gladman's implementation and our baseline implementation and it shows that both the key expansion and the encryption process of our baseline implementation is only marginally (1.5 times) slower than Gladman's implementation.

5.2 Comparison

Our Implementation. In Table 1, we present the speedup factor of our improved implementation (in Section 4) where the secure inversion in $GF(2^8)$ is computed over composite field, compared with our implementation of [7](given in Section 3) where we compute the secure inversion through exponentiation. Table 1 also shows the penalty factor due to the integration of high-order DPA countermeasures into our implementation (in Section 4) compared to the baseline implementation (given in Section 5.1). In the case of first-, second- and third-order, we use the highly-optimized "pure" NEON assembly implementation, which is approximately $(order + 1)^2$ times slower than the baseline implementation. In all other cases, we use the mixed C and NEON assembly (i.e. the generic) implementation, which is a little more than $(order + 1)^2$ times slower than the baseline implementation.

Table 1. Speedup factor of our improved implementation in Section 4 compared to Section 3, and penalty factor of our improved implementation compared to the baseline implementation

order	unmasked	1	2	3	4	5	6	7
Section 3	2,281	10,050	21,277	36,808	69,022	97,578	131,164	169,806
Section 4	*1,141*	4,869	9,127	14,855	34,875	47,640	61,915	77,820
Speedup Factor	2.0	2.1	2.3	2.5	2.0	2.0	2.1	2.2
Penalty Factor	-	4.3	8.0	13.0	30.6	41.8	54.3	68.2

A Note on Random Numbers We used a Pseudorandom Number Generator (PRNG) based on Linear Feedback Shift Register (LFSR). However most of the LFSR based PRNG are not cryptographically secure. To avoid this, one can replace the PRNG used with a stream cipher such as Salsa20.[11] The vectorized implementation of Salsa20 requires only 5.6 cycles/byte [2] and hence do not significantly impact our results.

Related Works. Here, we compare our implementation results with different countermeasures from CHES 2010 [18], CHES 2011 [14] and Eurocrypt 2014 [6]. However, in the original papers, the implementations were evaluated on platforms that are completely different from ARM NEON, which means these comparisons have to be taken with a pinch of salt. For example, the implementation reported in [14] was written in C and evaluated on an 8-bit ATmega128 processor, while Coron's [6] implementation was in C on a MacBook Air with a 32-bit Intel processor. Therefore, it makes only sense to compare the penalty factor of the different implementations for a given orders (see Table 2), but not the absolute execution times. Table 2 shows that our results are significantly better than that of the others. With our proposed implementation, the second and third order secure AES is only 8 and 13 times slower than the unmasked implementation. Moreover, our results achieve a speedup factor of three compared with the fastest solutions available.

Table 2. Penalty factor in different implementations

Method	Platform	first-order	second-order	third-order	fourth-order
CHES'10 [18]	8-bit 8051	65	132	235	-
CHES'11 [14]	8-bit AVR	-	22	39	-
Coron [Cor14]	1.84 GHz Intel	439	1205	2411	4003
Our (Section 3)	32-bit ARM	9	19	32	60
Our (Section 4)	32-bit ARM	4	8	13	31

[11] For further improving the security of random numbers, one could also use True Random Number Generator (TRNG) to seed the PRNG.

6 Conclusion

We addressed the efficiency problem of higher-order masking schemes for the AES. All current schemes described in the literature are extremely slow and, hence, little attractive for use in real-world applications. We found that the performance-critical parts in the implementation of these schemes are the field multiplication and random number generation. By using various algorithmic techniques (e.g. Barrett reduction and composite field arithmetic) and implementation improvements (i.e. efficient vectorization), we managed to reduce the overhead to a level that is more acceptable in practice. In particular, our NEON implementation of second and third-order secure AES is only eight and thirteen times slower than an unmasked baseline implementation that is resistant against cache-timing attacks. The presented higher-order masking is not only relevant for the AES, but also for various AES-based authenticated encryption schemes, which are currently evaluated in the CAESAR competition.

Acknowlegements

This work is partially supported by the National Natural Science Foundation of China under Grant No. 61173139.

References

1. P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer Berlin Heidelberg, 1987.
2. D. J. Bernstein and P. Schwabe. NEON crypto. In *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, pages 320–339, 2012.
3. T. Caddy. Differential Power Analysis. In H. C. van Tilborg and S. Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 336–338. Springer Verlag, 2011.
4. S. Chari, C. Jutla, J. R. Rao, and P. Rohatgi. A Cautionary Note Regarding Evaluation of AES Candidates on Smart-cards. In *Second Advanced Encryption Standard Candidate Conference*, pages 133–147, 1999.
5. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. Wiener, editor, *Advances in Cryptology CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer Berlin Heidelberg, 1999.
6. J.-S. Coron. Higher Order Masking of Look-Up Tables. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 441–458. Springer Berlin Heidelberg, 2014.
7. J.-S. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-Order Side Channel Security and Mask Refreshing. In S. Moriai, editor, *Fast Software Encryption*, Lecture Notes in Computer Science, pages 410–424. Springer Berlin Heidelberg, 2013.

8. J.-F. Dhem. Efficient Modular Reduction Algorithm in $\mathbb{F}_q[x]$ and Its Application to "Left to Right" Modular Multiplication in $\mathbb{F}_2[x]$. In C. D. Walter, c. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 203–213. Springer Berlin Heidelberg, 2003.
9. B. R. Gladman. AES and combined encryption/authentication modes. Available for download at http://gladman.plushost.co.uk/oldsite/AES/index.php, June 2006.
10. V. Grosso, F. Standaert, and S. Faust. Masking vs. Multiparty Computation: How Large Is the Gap for AES? In *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, pages 400–416, 2013.
11. V. Grosso, F. Standaert, and S. Faust. Masking vs. multiparty computation: how large is the gap for AES? *J. Cryptographic Engineering*, 4(1):47–57, 2014.
12. J. Guajardo and C. Paar. Efficient algorithms for elliptic curve cryptosystems. In J. Kaliski, BurtonS., editor, *Advances in Cryptology CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 342–356. Springer Berlin Heidelberg, 1997.
13. Y. Ishai, A. Sahai, and D. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer Berlin Heidelberg, 2003.
14. H. Kim, S. Hong, and J. Lim. A Fast and Provably Secure Higher-Order Masking of AES S-Box. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 95–107. Springer Berlin Heidelberg, 2011.
15. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer Berlin Heidelberg, 1999.
16. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, volume 31. Springer, 2008.
17. T. S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. In c. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 238–251. Springer Berlin Heidelberg, 2000.
18. M. Rivain and E. Prouff. Provably Secure Higher-Order Masking of AES. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer Berlin Heidelberg, 2010.
19. A. Rudra, P. Dubey, C. Jutla, V. Kumar, J. Rao, and P. Rohatgi. Efficient Rijndael Encryption Implementation with Composite Field Arithmetic. In c. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 171–184. Springer Berlin Heidelberg, 2001.
20. A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In C. Boyd, editor, *Advances in Cryptology ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer Berlin Heidelberg, 2001.
21. J. Waddle and D. Wagner. Towards Efficient Second-Order Power Analysis. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Sys-*

tems - CHES 2004, volume 3156 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2004.

A Algorithm for Secure Exponentiation in $GF(2^8)$

Algorithm 5 SecExp254 - Masked exponentiation to 254 over $GF(2^8)$ with n shares [18]

```
Input: shares $x_i$ satisfying $x_1 \oplus \cdots \oplus x_n = x$
Output: shares $y_i$ satisfying $y_1 \oplus \cdots \oplus y_n = x^{254}$
 1: for $i = 1$ to $n$ do
 2:     $z_i \leftarrow x_i^2$                                              $\triangleright \bigoplus_i z_i = x^2$
 3: end for
 4: RefreshMasks$(z_1, \cdots, z_n)$
 5: $(y_1, \cdots, y_n) \leftarrow$ SecMult$((z_1, \cdots, z_n), (x_1, \cdots, x_n))$    $\triangleright \bigoplus_i y_i = x^3$
 6: for $i = 1$ to $n$ do
 7:     $w_i \leftarrow y_i^4$                                              $\triangleright \bigoplus_i w_i = x^{12}$
 8: end for
 9: RefreshMasks$(w_1, \cdots, w_n)$
10: $(y_1, \cdots, y_n) \leftarrow$ SecMult$((y_1, \cdots, y_n), (w_1, \cdots, w_n))$    $\triangleright \bigoplus_i y_i = x^{15}$
11: for $i = 1$ to $n$ do
12:     $y_i \leftarrow y_i^{16}$                                             $\triangleright \bigoplus_i y_i = x^{240}$
13: end for
14: $(y_1, \cdots, y_n) \leftarrow$ SecMult$((y_1, \cdots, y_n), (w_1, \cdots, w_n))$    $\triangleright \bigoplus_i y_i = x^{252}$
15: $(y_1, \cdots, y_n) \leftarrow$ SecMult$((y_1, \cdots, y_n), (z_1, \cdots, z_n))$    $\triangleright \bigoplus_i y_i = x^{254}$
16: return $y_1, \cdots, y_n$
```

B Algorithm for Field Multiplication in $GF(2^4)$

Algorithm 6 Field multiplication in $GF(2^4)$

Input: Polynomials $A(x)$ and $B(x)$ in $GF(2^4)$
Output: Polynomial $Z(x) = A(x) \cdot B(x)$ `mod` $N(x)$, where $N(x) = x^4 + x + 1$.

Pre-computation:

```
1: $p \leftarrow deg(N(x))$                                        $\triangleright\ p = 4$
2: $\alpha \leftarrow 2 * (p - 1)$                                 $\triangleright\ \alpha = 6$
3: $\beta \geq \alpha - p$                                          $\triangleright\ \beta \geq 2$
4: $R(x) \leftarrow \lfloor \frac{x^{p+\beta}}{N(x)} \rfloor$      $\triangleright\ R(x) = x^2$ if $\beta = 2$
```

Multiplication with Barrett modular reduction(Theorem 1):

```
1: $U(x) \leftarrow A(x) \cdot B(x)$                               $\triangleright\ deg(U(x)) \leq 14$
2: $T(x) \leftarrow \lfloor \frac{U(x)}{x^p} \rfloor$               $\triangleright\ deg(T(x)) \leq 6$ and $Q(x) = T(x)$
3: $V(x) \leftarrow Q(x) \cdot N(x)$                               $\triangleright\ deg(V(x)) \leq 14$
4: $Z(x) \leftarrow U(x) + V(x)$
```