IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 2 March 2023, accepted 11 April 2023, date of publication 19 April 2023, date of current version 26 April 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3268642

RESEARCH ARTICLE

Side-Channel Resistant 2048-Bit RSA Implementation for Wireless Sensor Networks and Internet of Things

UTKU GULEN[1] AND SELCUK BAKTIR[2], (Member, IEEE)
[1]Department of Computer Engineering, Faculty of Engineering and Natural Sciences, Bahcesehir University, 34353 Istanbul, Turkey
[2]College of Engineering and Technology, American University of the Middle East, Egaila 54200, Kuwait

Corresponding author: Selcuk Baktir (selcuk.baktir@aum.edu.kw)

ABSTRACT We present a practical realization of Rivest-Shamir-Adleman (RSA) with a 2048-bit key on MSP430, a widely used microcontroller in wireless sensor network and Internet of things applications, and show that 2048-bit RSA is feasible on a constrained microcontroller. We exploit several methods for acceleration, e.g. Montgomery modular multiplication, subtractive Karatsuba-Ofman and Chinese remainder theorem (CRT) based modular exponentiation, and achieve RSA encryption and decryption with a 2048-bit key on MSP430 in just 0.14 s and 7.56 s, respectively. Our implementation on the low-end MSP430 microcontroller achieves 2048-bit RSA significantly faster ($\times$2.9 and $\times$2.4 for encryption and decryption) with respect to the existing implementation in the literature on the comparable ATmega128 microcontroller. While our implementation is secure against the brute force attack due to its 2048-bit key, and thus 112-bit security level, it also includes the necessary side-channel countermeasures, e. g. message and key blinding, to help mitigate implementation attacks such as simple power analysis and differential power analysis.

INDEX TERMS CRT exponentiation, embedded systems, Internet of Things, Karatsuba-Ofman, public-key cryptography, Rivest-Shamir-Adelman, RSA, side-channel resistant, wireless sensor networks.

I. INTRODUCTION

It is important to implement cryptographic algorithms efficiently on low-end microcontrollers to provide security in a growing number of wireless sensor network (WSN) and Internet of things (IoT) applications [1], [2], [3]. These applications transmit and/or process sensitive data in areas such as smart homes, connected cars, smart grids, smart healthcare systems, smart farming [4], environmental surveillance [5], smart cities, smart factories [6] and military applications [7], [8], [9]. While symmetric key cryptographic algorithms would facilitate secure communication for these applications at low cost, distribution of the secret key could be achieved most practically using complex public-key cryptography (PKC) algorithms.

The Rivest-Shamir-Adleman (RSA) cryptosystem, which is the first general-purpose PKC algorithm, is by far also the most widely deployed one [2], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. Nevertheless, memory and CPU speed limitations for low-end microcontrollers make it challenging to implement RSA on constrained microcontrollers used in WSNs and IoT systems [22]. For the 80-bit security level, RSA should use a 1024-bit key. However, as the 80-bit security level is considered out-of-date for most applications, the 112-bit security level, and hence the use of at least a 2048-bit key, is suggested for RSA [23]. While the same security level is reached with elliptic-curve cryptography (ECC) utilizing a shorter key and hence smaller computational load [24], [25], RSA is still the most widespread public-key cryptographic algorithm. RSA has some advantages over ECC. One advantage is signature verification with RSA is faster. Furthermore, RSA is more mature and more widely adopted, especially in Internet applications. Any WSN or IoT application that uses RSA would have a better chance of being compatible with existing infrastructures. Finally, while the prospect of building a general-purpose quantum

The associate editor coordinating the review of this manuscript and approving it for publication was Dominik Strzalka.

computer would undermine the security of both RSA and ECC, ECC has also been the suspect of a more recent threat which is the potential back doors due to its parameter-based nature as revealed by Edward Snowden [26]. Hence, RSA clearly has some strong points against ECC.

The National Institute of Standards Technology (NIST) recommends the use of at least a 2048-bit long RSA key to achieve 112-bit security [23]. Nevertheless, to the best of our knowledge, there is no existing study that efficiently implements RSA with a 2048-bit or longer key on a constrained microcontroller such as MSP430 and ATmega. Texas Instrument's low-cost and low-power family of MP430 microcontrollers are some of the most common microcontrollers which are used in wireless sensor nodes. For instance, the well-known sensor nodes Telos, Tmote and BEAN use the MSP430F149 microcontroller; moreover, the TelosB, Tmote Sky, KMote and SHIMMER sensor nodes use the MSP430F1611 microcontroller [27]. Similarly, the sensor nodes WSN1120L, WSN1120CL, WSN1101ANL and WSN1101ACL of WiSense use the MSP430G2955 microcontroller [28], [29], [30], [31]. In our work, we implement 2048-bit RSA with the necessary mitigation techniques against side-channel attacks for the MSP430 family of microcontrollers. We combine several acceleration methods together. Our 2048-bit RSA implementation with side-channel protection has by far the fastest timing result in the literature on a constrained microcontroller such as MSP430 or ATmega128.

A. OUR MAIN CONTRIBUTIONS

- For the first time in the literature, we present efficient RSA implementations on the MSP430 family of constrained microcontrollers using a 2048-bit key as recommended by the NIST [23].
- We combine the acceleration techniques sliding-window method, Chinese remainder theorem (CRT) based exponentiation, Montgomery modular multiplication, and subtractive Karatsuba-Ofman multiplication for the first time in the literature for 2048-bit arithmetic. Our resulting 2048-bit RSA implementation on the constrained MSP430 microcontroller outperforms the existing implementation on the comparable ATmega128 microcontroller [32], and achieves RSA encryption and decryption operations more than twice faster.
- Unlike the existing work in [32], our 2048-bit RSA implementation includes the necessary countermeasures, such as constant time implementation, secret key blinding, ciphertext blinding and sliding window, to prevent vulnerabilities that may arise from implementation attacks such as simple power analysis (SPA) and differential power analysis (DPA) [33].
- We show that strong RSA cryptography with a 2048-bit key is feasible on constrained microcontrollers used in WSN and IoT applications.

In Section II, we present preliminary information and foundations on RSA and arithmetic algorithms for accelerating it. We present our 2048-bit RSA implementations on three generations of the MSP430 microcontroller in Section III. Finally, Section IV includes the performance evaluations and comparisons with the existing work.

II. PRELIMINARIES

RSA [34], proposed in 1978, is the oldest as well as the most extensively deployed general purpose public-key cryptosystem [35]. Encryption and decryption operations with RSA are achieved simply with modular exponentiations. For the prime numbers p and q, modulus $N = p \cdot q$, private key d, public key e, cipher text C and plain text M, the RSA encryption and decryption operations are conducted as given below:

$$\text{RSA Encryption} : C = M^e \bmod N,$$
$$\text{RSA Decryption} : M = C^d \bmod N.$$

In the RSA algorithm, the private key d and the public key e are related to each other according to the following equality

$$e \cdot d = 1 \pmod{\phi(N)},$$

where the Euler's Phi function $\phi(N)$ is defined as

$$\phi(N) = (p-1) \cdot (q-1).$$

If one can factorize N, they would be able to easily compute $\phi(N)$ and thus the private key $d = e^{-1} \bmod \phi(N)$. Hence, the security of RSA depends on the difficulty of factorizing the RSA modulus N which is the product of the prime numbers p and q. Choosing larger values for p and q would make factorization of N harder and thus increase the security of RSA.

Several methods could be utilized to expedite the encryption/decryption processes in RSA. While the Chinese remainder theorem (CRT) based multiplication and the sliding window method could be used to accelerate RSA decryption, the small exponent $e = 2^{16} + 1$ could be used to accelerate RSA encryption. In order to efficiently realize modular exponentiation, conducted in RSA encryption and decryption, it is necessary to implement modular multiplication efficiently. Montgomery multiplication is a popular technique that is commonly used for efficient modular multiplication [36]. As part of Montgomery multiplication, one needs to achieve integer multiplications. Karatsuba-Ofman algorithm is an elegant and widely used technique to speed up integer multiplication [37]. The subtractive Karatsuba-Ofman technique improves upon the original Karatsuba-Ofman algorithm and can be used to achieve the integer multiplication operations required for Montgomery multiplication [38].

A. SLIDING WINDOW METHOD

Modular exponentiation with large integers, as conducted in RSA encryption/decryption, is considered feasible due to the binary method, aka the square-and-multiply algorithm [39]. In this method, the exponent bits are scanned and processed one at a time starting with the highest ordered nonzero bit. The intermediary result is first set to 1 for the highest ordered

Algorithm 1 Modular Exponentiation With the 4-Bit Sliding Window Method

Require: $B, N, k = (k_{t-1} \cdots k_1 k_0)_2$
Ensure: $B^k \bmod N$
1: $T_0 \leftarrow 1$
2: $T_1 \leftarrow B$
3: **for** $j \leftarrow 2 \rightarrow 2^4 - 1$ **do**
4: $\quad T_j \leftarrow T_{j-1} \cdot B$
5: **end for**
6: $R \leftarrow T_0$
7: $j \leftarrow 0$
8: **while** $j \leq t - 1$ **do**
9: $\quad i \leftarrow (k_{t-1-j} k_{t-2-j} k_{t-3-j} k_{t-4-j})_2$
10: $\quad$ **for** $l \leftarrow 1 \rightarrow 4$ **do**
11: $\quad\quad R \leftarrow R^2$
12: $\quad$ **end for**
13: $\quad R \leftarrow R \cdot T_i$
14: $\quad j \leftarrow j + 4$
15: **end while**
Return: (R)

nonzero bit and then squared for each newly scanned bit. If the scanned bit is 0, no further operations are performed for that bit. On the other hand, the intermediary result is further multiplied with the base if the scanned bit is 1. The exponentiation operation is finalized when all bits of the exponent are scanned and processed following this scheme. Although it makes exponentiation feasible, the binary method is considered inefficient and also known to be insecure against the SPA attack [33]. The binary method can be used safely in RSA encryption where a small public key is utilized to simplify exponentiation. With the small exponent $e = 2^{16} + 1$, the RSA encryption operation $C = M^e \bmod N$ can be computed utilizing the binary method by conducting only sixteen squarings and one multiplication modulo N.

An alternative to the binary method for exponentiation is the sliding window method where the exponent's bits are scanned two or more bits at once. The 4-bit sliding window method is given in Algorithm 1. In this method, the first 16 powers of the base, starting with the 0^{th} power, are precomputed and saved in a look-up table. The intermediary result is initially set to 1. Then, for every 4 bits of the exponent, starting with the highest ordered bits, the intermediary result is squared 4 times and then multiplied with the corresponding look-up table entry. Similar to the binary method, one squaring is performed for every scanned bit. However, only one multiplication is done for every 4-bit window of scanned bits, which significantly reduces the total number of required multiplications. Furthermore, unlike the binary method, the sliding window method is considered secure against the SPA attack. Hence, the sliding window method could be used to realize modular exponentiation for RSA decryption.

B. RSA DECRYPTION USING CRT BASED EXPONENTIATION

The CRT is commonly taken advantage of to accelerate RSA decryption. In RSA, the modulus N is equal to the product of the prime numbers p and q, therefore CRT based exponentiation modulo N can be facilitated to accelerate the exponentiation operation in RSA decryption. The decryption operation, $M = C^d \bmod N$, could be computed utilizing CRT based modular exponentiation as follows [40]:

$$\begin{aligned} C_p &= C \bmod p, \\ C_q &= C \bmod q, \\ M_p &= C_p^{d_p} \bmod p, \\ M_q &= C_q^{d_q} \bmod q, \\ M &= (q \cdot c_p) \cdot M_p + (p \cdot c_q) \cdot M_q \bmod N, \end{aligned}$$

where the constants d_p, d_q, c_p and c_q are precomputed as

$$\begin{aligned} d_p &= d \bmod (p-1), \\ d_q &= d \bmod (q-1), \\ c_p &= q^{-1} \bmod p, \\ c_q &= p^{-1} \bmod q. \end{aligned}$$

The lengths of the operands d_p, d_q, p, q, c_p and c_q are half the lengths for the corresponding operands N, d and C in normal RSA decryption. Thus, the arithmetic operations in CRT based RSA decryption are performed over smaller operands and hence they are simpler compared to the arithmetic operations in normal RSA decryption. A speedup by a factor of up to four is achieved in RSA decryption by utilizing CRT based exponentiation [40].

C. MONTGOMERY MULTIPLICATION

Montgomery multiplication is an elegant technique for modular multiplication. It is typically used in algorithms where repeated modular multiplications are performed, such as the modular exponentiation in RSA encryption and decryption operations [36], [41]. Algorithm 2 gives the steps of Montgomery multiplication for multiplying the n-bit integers A and B modulo the n-bit modulus N. In the algorithm, the operands are in their Montgomery forms $\bar{A} = A \cdot r \bmod N$ and $\bar{B} = B \cdot r \bmod N$ where $r = 2^n$. Note in Algorithm 2 that an n-bit Montgomery multiplication is achieved with 3 n-bit integer multiplications, one $2n$-bit addition, and one n-bit subtraction. The cost of division by r in the third line of Algorithm 2 is negligible since the result of $T_2 \cdot N$ is a multiple of $r = 2^n$. Thus, $T_2 \cdot N/r$ can be obtained simply by discarding the least significant n bits of $T_2 \cdot N$. In line 5 of Algorithm 2, the subtraction operation depends on the if condition. In our work, we make this if statement execute in constant time to provide side-channel resistance. We explain the side-channel attack countermeasures in our RSA implementation in detail in Section III-B.

D. SUBTRACTIVE KARATSUBA-OFMAN TECHNIQUE

Multiprecision integer multiplication, which is performed three times in Montgomery multiplication, is the core arithmetic operation in RSA. Therefore achieving fast multiprecision integer multiplication is crucial for an efficient RSA implementation. Using the classical grade school method,

Algorithm 2 Montgomery Modular Multiplication

Require: $\bar{A} = A \cdot r \bmod N$ **and** $\bar{B} = B \cdot r \bmod N$
where $r = 2^n$ and $n = \lceil \log_2 N \rceil$
Ensure: $\bar{C} = A \cdot B \cdot r \bmod N$
1: $T_1 \leftarrow \bar{A} \cdot \bar{B}$
2: $T_2 \leftarrow T_1 \cdot N' \bmod r$ **where** $N' = -N^{-1} \bmod r$
3: $T_1 \leftarrow (T_1 + T_2 \cdot N)/r$
4: **if** $T_1 \geq N$ **then**
5: $T_1 \leftarrow T_1 - N$
6: **end if**
Return: $(\bar{Z} \leftarrow T_1)$

multiplication of large integers is typically achieved in terms of a large number of word additions and multiplications. The Karatsuba-Ofman technique is a divide-and-conquer method that trades computationally expensive multiplications with simple additions [37]. An illustration of Karatsuba-Ofman for multiplying the n-bit integers A and B is given in Figure 1. As given with Figure 1, in Karatsuba-Ofman, A and B are bisected to their higher and lower ordered parts, denoted with A_H, A_L and B_H, B_L, respectively, and arithmetic is performed over these operand halves. Thus, an n-bit multiplication is achieved with roughly three $\frac{n}{2}$-bit multiplications and a number of additions/subtractions. Since multiplication is more complex and takes more time compared to addition/subtraction, Karatsuba-Ofman ensures faster overall multiplication. Note that while the classical grade school method requires performing four $\frac{n}{2}$-bit multiplications to achieve an n-bit multiplication, Karatsuba-Ofman requires only three. When applied recursively, Karatsuba-Ofman significantly reduces the complexity of multiprecision multiplication from $\mathcal{O}(m^2)$ to $\mathcal{O}(m^{log_2 3})$ for the multiplication of two m word integers.

In the Karatsuba-Ofman technique, since additions may generate carry bits, the multiplication operation $(A_H + A_L) \cdot (B_H + B_L)$ is not always fixed in size. When the conditional branch operation is avoided in this multiplication computation to prevent timing attacks, the operands $A_H + A_L$ and $B_H + B_L$ are considered with their extra carry bit even when a carry bit is not generated after the addition. This causes overhead in the multiplication computation. This overhead can be eliminated by using the *subtractive Karatsuba-Ofman* technique, a slightly optimized form of the original Karatsuba-Ofman [38], [42]. With this approach, the two halves of the integers to be multiplied are subtracted from each other, instead of being added. Figure 2 shows the operations that take place in subtractive Karatsuba-Ofman.

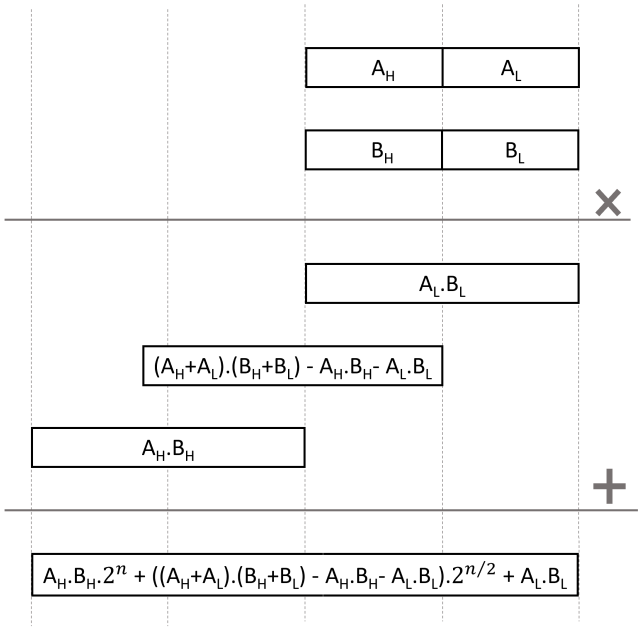


FIGURE 1. Karatsuba-Ofman multiplication for computing $A \times B$.

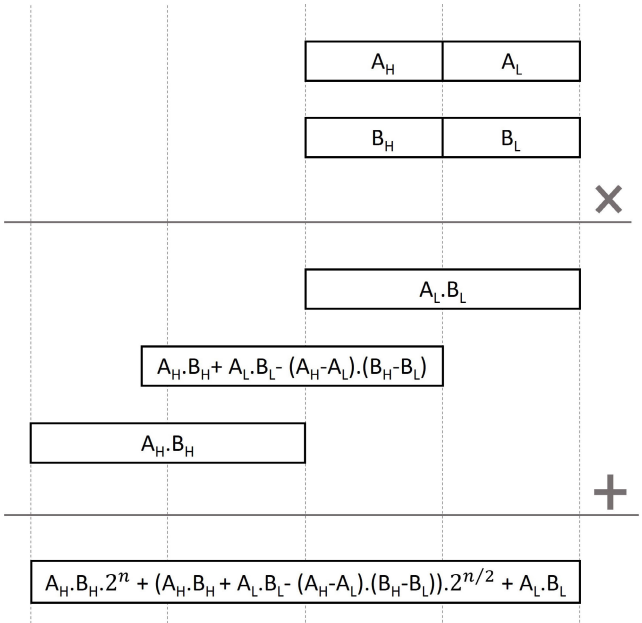


FIGURE 2. Subtractive Karatsuba-Ofman multiplication for computing $A \times B$.

III. OUR 2048-BIT RSA IMPLEMENTATIONS

We implement 2048-bit RSA on three target MSP430 microcontrollers, namely MSP430F1611 [45], MS430F2618 [44] and MSP430F5529 [43]. Since these three generations of MSP430 support the same instruction set, we are able to use the same code with little modifications for the three microcontrollers. However, there are some differences between our target MSP430 microcontrollers. For instance, MSP430F5529 has a 32 × 32 multiplier whereas MSP430F2618 and MSP430F1611 have a 16 × 16 multiplier, and hence word multiplications need to be handled differently. Other than the size of the hardware multiplier, the memory write instruction takes different numbers of clock cycles on different versions of MSP430. The memory write instruction on MSP430F1611 takes 4 clock cycles while the same instruction takes 3 clock cycles on MSP430F2618 and MDP430F5529. Another difference between the microcontrollers is in their memory capacities and their maximum CPU clock frequencies. MSP430F5529, MSP430F2618 and

MSP430F1611 have the memory sizes of 128 kB, 116 kB and 48 kB, and the maximum CPU clock frequencies of 25 MHz, 16 MHz and 8 Mhz, respectively.

We use the IAR Embedded Workbench development environment, and test our code using its debugger and clock-counter features [46]. All the acceleration techniques described in Section II are applied in our implementations, namely CRT-based exponentiation and 4-bit sliding window to accelerate RSA decryption, the small exponent $2^{16} + 1$ to accelerate RSA encryption, and subtractive Karatsuba-Ofman and Montgomery multiplication to accelerate both. For core arithmetic operations such as integer addition/subtraction and subtractive Karatsuba-Ofman, we write our codes in assembly to guarantee fast execution.

In our assembly codes, we eliminate conditional branch and jump instructions to mitigate timing attacks. For MSP430F1611 and MS430F2618, we implement 2048-bit integer multiplication by recursively utilizing subtractive Karatsuba-Ofman (for five levels) until the method does not accelerate the base integer multiplication any further. Here, the base integer multiplications, which are 64-bit multiplications, are implemented using the onboard 16×16 multiplier. For MSP430F5529 with a 32×32 multiplier, we recursively implement subtractive Karatsuba-Ofman (for four levels) until the 128-bit multiplications are reached at the base case of recursion. We observe that subtractive Karatsuba-Ofman does not speed up 128-bit integer multiplication when the 32×32 multiplier is used for word multiplications. While we implement subtractive Karatsuba-Ofman recursively, our code is fully unrolled and thus there is no timing overhead due to recursive function calls.

We optimize our integer arithmetic as much as we can, e.g. by storing frequently used operands in registers, to reduce memory read/write overheads and optimize our subtractive Karatsuba-Ofman code for the squaring operation. We explain our acceleration optimizations for 128-bit subtractive Karatsuba-Ofman by giving assembly code examples with MSP430 instructions in Section III-A

The techniques which we use in our RSA decryption/encryption implementation are summarized in Figure 3. At the bottom of our implementation, fast integer multiplication/squaring is achieved with the *subtractive Karatsuba-Ofman* and *operand scanning* methods which are used within *Montgomery multiplication*. Montgomery multiplication is used in *CRT based modular exponentiation* and in modular exponentiation with *small public exponent*. CRT-based modular exponentiation is used in RSA decryption and modular exponentiation with small public exponent is used in RSA encryption. Furthermore, in RSA decryption, *message and key blinding* techniques, which we describe in Section III-B, are used to mitigate side-channel attacks and protect the private decryption key. Finally, the sliding window method is used in RSA decryption for both side-channel attack protection and efficiency.

A. OPTIMIZED ARITHMETIC FOR SUBTRACTIVE KARATSUBA-OFMAN AND OPERAND SCANNING

1) SUBTRACTIVE KARATSUBA-OFMAN MULTIPLICATION IN FIXED-TIME

At the bottom of our 2048-bit recursive subtractive Karatsuba-Ofman implementation on MSP430F1611 and MSP430F2618, we perform 128-bit subtractive Karatsuba-Ofman. Here, we explain the details of our 128-bit subtractive Karatsuba-Ofman implementation and the optimizations we use. We would like to note that we apply the same techniques at the higher levels of recursion in our 2048-bit recursive subtractive Karatsuba-Ofman implementation. As described in Figure 2, for performing 128-bit subtractive Karatsuba-Ofman, three 64-bit multiplications are performed, namely $A_H \cdot B_H$, $(A_H - A_L) \cdot (B_H - B_L)$ and $A_L \cdot B_L$. The advantage of subtractive Karatsuba-Ofman over normal Karatsuba-Ofman is that the multiplication operation $(A_H - A_L) \cdot (B_H - B_L)$ is fixed in size since there is no possibility of a carry occurrence in the computations of $A_H - A_L$ and $B_H - B_L$, unlike in the additions $A_H + A_L$ and $B_H + B_L$ that take place in the original Karatsuba-Ofman method. The multiplication operation $(A_H - A_L) \cdot (B_H - B_L)$ in subtractive Karatsuba-Ofman is performed over shorter operands and hence it is more efficient compared to the multiplication operation $(A_H + A_L) \cdot (B_H + B_L)$ that takes place in the original Karatsuba-Ofman algorithm.

We use the two's complement representation to store the results of the subtractions $A_H - A_L$ and $B_H - B_L$. We use an additional sign word that is set to $0 \times$ FFFF or $0 \times$ 0000 depending on whether the result of the subtraction is negative or positive. We denote the sign words for the results of $A_H - A_L$ and $B_H - B_L$ with SWA and SWB, respectively. In order to avoid timing attacks, we realize the subtractions $A_H - A_L$ and $B_H - B_L$, as well as the multiplication $(A_H - A_L) \cdot (B_H - B_L)$, in fixed execution time and without using branches, regardless of whether the result is positive or negative. We obtain and process the magnitudes of the results of the subtraction operations. In order to do this, we need to compute the two's complement of the result if subtraction results in a negative number. We do this computation in fixed time, regardless of whether a subtraction results in a positive or a negative number, by XORing the sign word with all the remaining words of the result and then by adding the sign bit, as shown in Subroutines 1 and 2. Note that the sign words SWA and SWB will be either $0 \times$FFFF or 0×0000, depending on whether the results of $A_H - A_L$ and $B_H - B_L$ are negative or positive, respectively. Hence, XORing $A_H - A_L$ (or $B_H - B_L$) with its sign word and then adding the sign bit to the result would give us its two's complement, and thus its magnitude, only if its sign word is $0 \times$ FFFF (it is initially negative). If $A_H - A_L$ (or $B_H - B_L$) is positive, and hence its sign word is 0×0000, this operation will not change its value which is already the positive magnitude. Note that, in Subroutines 1 and 2, the magnitudes of $A_H - A_L$ and $B_H - B_L$ are computed and stored in the arrays $A[4{:}7]$ and $B[4{:}7]$, and their sign words are stored in SWA and SWB, respectively.

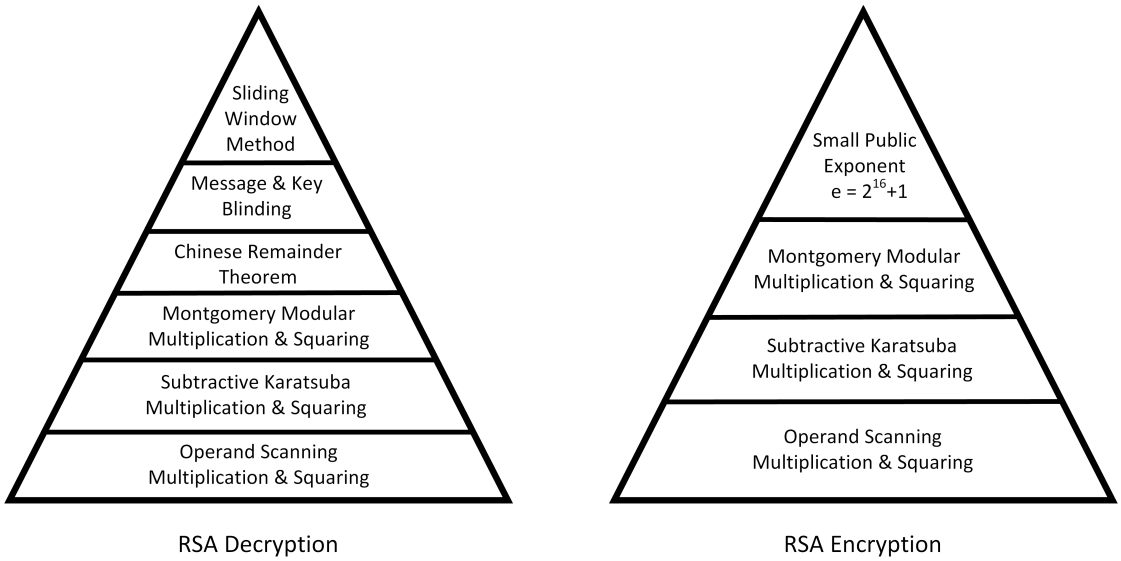


FIGURE 3. Techniques used in the proposed RSA implementation listed according to their performance impact.

```
CLR SWA
SUBC A[0],A[4]
SUBC A[1],A[5]
SUBC A[2],A[6]
SUBC A[3],A[7]
SBC SWA
XOR SWA,A[4]
XOR SWA,A[5]
XOR SWA,A[6]
XOR SWA,A[7]
RRA SWA
ADC A[4]
ADC A[5]
ADC A[6]
ADC A[7]
```

Subroutine 1. Assembly subroutine for A_H-A_L computation in 128-bit subtractive Karatsuba multiplication.

```
CLR SWB
SUBC B[0],B[4]
SUBC B[1],B[5]
SUBC B[2],B[6]
SUBC B[3],B[7]
SBC SWB
XOR SWB,B[4]
XOR SWB,B[5]
XOR SWB,B[6]
XOR SWB,B[7]
RRA SWB
ADC B[4]
ADC B[5]
ADC B[6]
ADC B[7]
```

Subroutine 2. Assembly subroutine for B_H-B_L computation in 128-bit subtractive Karatsuba multiplication.

We use the sign words of $A_H - A_L$ and $B_H - B_L$, stored in SWA and SWB, in the computation of the intermediary product $-T_1 = (A_H - A_L) \cdot (B_H - B_L)$ which can be positive or negative. By utilizing the sign words SWA and SWB, we compute $-T_1$ and add it to $T_2 = A_H \cdot B_H$ and $T_0 = A_L \cdot B_L$, as depicted in Figure 2. With Subroutine 3, we give our assembly code implementation for the computation of $T_0 + T_2 - T_1$ in 128-bit subtractive Karatsuba-Ofman. Here, firstly the computation of $-T_1$ is achieved, and then it is added with $T_2 = A_H \cdot B_H$ and $T_0 = A_L \cdot B_L$. Note that, in the beginning of Subroutine 3, the magnitudes of T_2, T_1 and T_0 are stored in the memory arrays $T2[0:7]$, $T1[0:7]$ and $T0[0:7]$, respectively. Remember that the sign words of $A_H - A_L$ and $B_H - B_L$ are stored in SWA and SWB at this point. After the subroutine is executed, the result $T_2 + T_0 - T_1$ is stored in the memory array $T1[0:7]$ and the carry-out bit is stored in SWA.

As seen in Figure 2, the lower half of $T_0 = A_L \cdot B_L$ gives us the least significant 64 bits of the result. To finalize 128-bit subtractive Karatsuba-Ofman multiplication, we add the upper half of T_0 to the lower half of $T_2 + T_0 - T_1$ which gives us the following 64 bits of the result. Finally, we add the generated carry bit and $T_2 = A_H \cdot B_H$ to the upper half of $T_2 + T_0 - T_1$ to generate the most significant 128 bits of the result. Subroutine 4 shows the assembly code for this summation operation where $T_2 = A_H \cdot B_H$ and $T_0 = A_L \cdot B_L$ are stored in the memory arrays $T2[0:7]$ and $T0[0:7]$, respectively.

Note that, we give with Subroutines 1-4 the assembly codes for 128-bit subtractive Karatsuba-Ofman. For 256-bit, 512-bit, 1024-bit and 2048-bit subtractive Karatsuba, we expand our codes by applying the same techniques recursively and in an unrolled fashion.

2) SUBTRACTIVE KARATSUBA-OFMAN SQUARING IN FIXED-TIME

We perform the modular exponentiation operations required for RSA decryption by using the 4-bit sliding window technique given with Algorithm 1. With this technique, four modular squarings are performed for every modular multiplication. Since the number of performed modular squarings is four times higher than modular multiplications, it is particularly important to improve the performance of modular squaring for fast RSA decryption. Similarly, in RSA encryption where the short public key $e = 2^{16} + 1$ is used for speed,

```
XOR SWB,SWA
XOR 0xFFFF,SWA
XOR SWA,T1[0]
XOR SWA,T1[1]
XOR SWA,T1[2]
XOR SWA,T1[3]
XOR SWA,T1[4]
XOR SWA,T1[5]
XOR SWA,T1[6]
XOR SWA,T1[7]
RRA SWA
ADDC T0[0],T1[0]
ADDC T0[1],T1[1]
ADDC T0[2],T1[2]
ADDC T0[3],T1[3]
ADDC T0[4],T1[4]
ADDC T0[5],T1[5]
ADDC T0[6],T1[6]
ADDC T0[7],T1[7]
ADC SWA
ADDC T2[0],T1[0]
ADDC T2[1],T1[1]
ADDC T2[2],T1[2]
ADDC T2[3],T1[3]
ADDC T2[4],T1[4]
ADDC T2[5],T1[5]
ADDC T2[6],T1[6]
ADDC T2[7],T1[7]
ADC SWA
```

Subroutine 3. Assembly subroutine for $t_2 + t_0 - t_1$ computation in 128-bit subtractive Karatsuba multiplication.

```
ADD T1[0],T0[4]
ADDC T1[1],T0[5]
ADDC T1[2],T0[6]
ADDC T1[3],T0[7]
ADDC T1[4],T2[0]
ADDC T1[5],T2[1]
ADDC T1[6],T2[2]
ADDC T1[7],T2[3]
ADDC SWA,T2[4]
ADC T2[5]
ADC T2[6]
ADC T2[7]
```

Subroutine 4. Subroutine for adding $t_2 + t_0 - t_1$ to finalize 128-bit subtractive Karatsuba multiplication.

encryption is achieved by performing 16 modular squarings and only one modular multiplication, and hence speeding up the modular squaring operation would pay off. Remember that we perform modular multiplication, as well as modular squaring, by using Montgomery multiplication given with Algorithm 2. In Montgomery multiplication, three integer multiplications are performed, as shown in lines 1, 2 and 3 of Algorithm 2. The only difference between modular multiplication and modular squaring with Montgomery multiplication is that in modular squaring the integer multiplication in line 1 of Algorithm 2 is a squaring. We speed up this integer squaring by optimizing our subtractive Karatsuba-Ofman multiplication implementation for the squaring computation.

The operations that take place in our subtractive Karatsuba-Ofman squaring implementation are depicted in Figure 4. Subtractive Karatsuba-Ofman squaring is faster than subtractive Karatsuba-Ofman multiplication since integer squaring is performed over a single operand and therefore the number of required memory read/write operations is less since the same values are multiplied. Moreover, in the computation of $T_1 = (A_H - A_L)^2$, the result is always

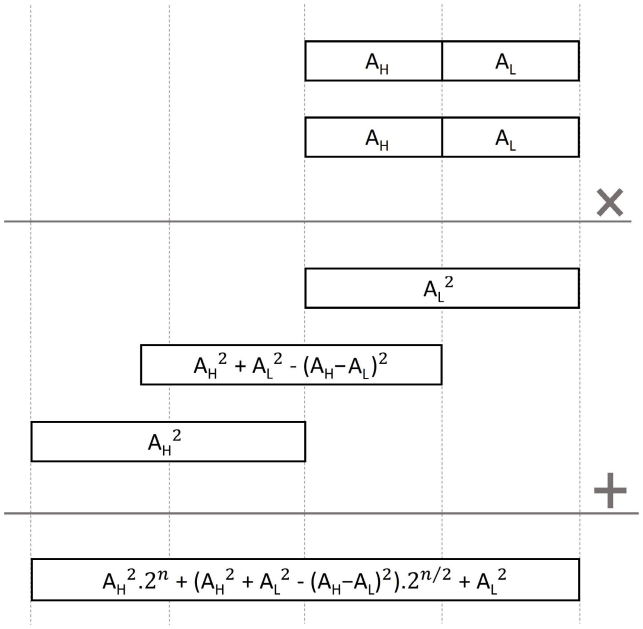


FIGURE 4. Subtractive Karatsuba-Ofman squaring for computing A^2.

positive which allows us to eliminate the XOR and two's complement operations that would otherwise be needed to handle the sign word. For 128-bit subtractive Karatsuba-Ofman squaring, as seen in Figure 4, the lower half of $T_0 = A_L^2$ gives us the least significant 64 bits of the result. After squaring $A_H - A_L$ to obtain $T_1 = (A_H - A_L)^2$, we subtract $T_0 = A_L^2$ and $T_2 = A_H^2$ from it. We give the assembly code for this operation in Subroutine 5 where T_0, T_1 and T_2 are stored in the memory arrays $T0[0:7]$, $T1[0:7]$ and $T2[0:7]$, respectively. Note that after this computation, the resulting value of $T_1 - T_0 - T_2 = (A_H - A_L)^2 - A_L^2 - A_H^2$ resides in the memory array $T1[0:7]$ and the sign word CW. We represent this result as $T1[0:7] \,||\, CW$, the concatenation of $T1[0:7]$ and CW. We then subtract $T_1 - T_0 - T_2$, stored in $T1[0:7] \,||\, CW$, from $T0[4:7] \,||\, T2[0:4]$ in order to compute the middle 128 bits of the result. We add the generated carry word to $T2[5:7]$ to compute the most significant 64 bits of the result and finalize the subtractive Karatsuba-Ofman squaring operation. When 128-bit subtractive Karatsuba-Ofman squaring completes execution, the lower 128 bits of the 256-bit result are stored in the memory array $T0[0:7]$ and the higher 128 bits are stored in the memory array $T2[0:7]$. We give our assembly code for this operation with Subroutine 6.

3) OPERAND-SCANNING MULTIPLICATION AND SQUARING IN FIXED-TIME

In our 2048-bit integer multiplication and squaring implementations on MSP430F1611 and MSP430F2618, after five levels of recursive subtractive Karatsuba-Ofman multiplication/squaring operations, at the base case of recursion we realize 64-bit multiplication/squaring operations using the operand scanning method and the onboard 16 × 16-bit hardware multiplier.

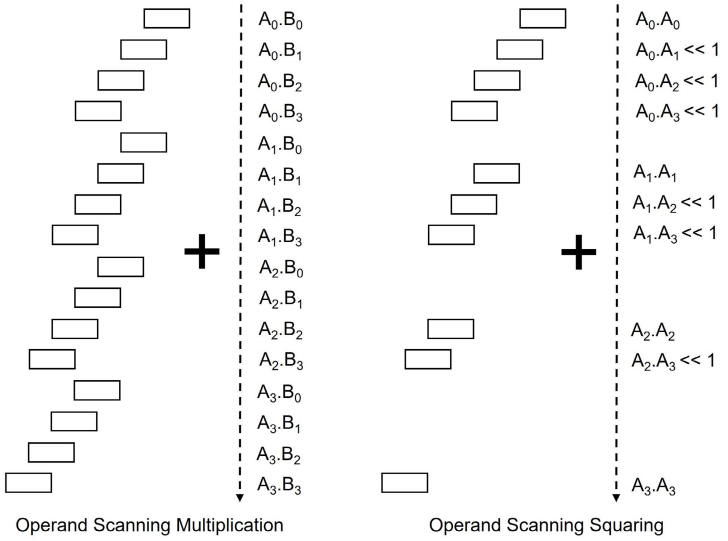


FIGURE 5. 64-bit operand scanning multiplication and squaring methods using a 16 × 16-bit hardware multiplier.

For our implementation on MSP430F5529, which has a 32 × 32-bit onboard hardware multiplier, we implement subtractive Karatsuba-Ofman with four levels of recursion. At the base case of recursion, we realize 128-bit multiplication/squaring operations using the operand scanning method and the onboard 32 × 32-bit hardware multiplier.

We do not carry out subtractive Karatsuba-Ofman fully recursively until we reach the microcontroller's word size, because the operand scanning method performs better than subtractive Karatsuba-Ofman for 64-bit operands. In the operand scanning method with 64-bit operands, we are able to store the partial products of word multiplications using only the 12 general purpose registers. We read and write the intermediary results by using these registers instead doing costly memory read/write operations. Subtractive Karatsuba-Ofman generates partial products with different sizes, i.e. $T_2 = A_H \cdot A_H$, $T_1 = (A_H - A_L) \cdot (B_H - B_L)$ and $T_0 = A_L \cdot A_L$, and requires irregular operand access patterns for computations with these partial products. Therefore, even in 64-bit subtractive Karatsuba-Ofman multiplication, memory read/write operations are inevitable in addition to register operations, and hence more clock cycles are spent compared to the operand scanning method. That is why we use the operand scanning method for multiplications at the base case for our recursive subtractive Karatsuba-Ofman implementation. Note that, for MSP430F5520 with a 32 × 32-bit hardware multiplier, we similarly use the operand scanning method for the 128-bit multiplication operations at the base case of our 4-level recursive subtractive Karatsuba-Ofman implementation. We optimize our operand scanning multiplication implementation on MSP430F5520 to handle 128-bit operands efficiently by emptying and reusing registers to avoid memory read/write operations while processing partial results.

We optimize our 64-bit and 128-bit operand scanning multiplication implementations for the *squaring* computation on our target microcontrollers with the 16 × 16-bit and 32 × 32-bit hardware multipliers, respectively. In *operand scanning squaring*, we are able to reduce the number of required word multiplications, compared with operand scanning multiplication, by eliminating repeating word multiplications in partial product computations. For instance, in the computation of $A_1 \cdot A_2 + A_2 \cdot A_1$, we eliminate the second word multiplication and find the result with the computation $A_1 \cdot A_2 \ll 1$ where we simply do a bitwise left shift operation on the result of the first word multiplication. With

```
CLR CW
SUB T0[0],T1[0]
SUBC T0[1],T1[1]
SUBC T0[2],T1[2]
SUBC T0[3],T1[3]
SUBC T0[4],T1[4]
SUBC T0[5],T1[5]
SUBC T0[6],T1[6]
SUBC T0[7],T1[7]
SBC CW
SUBC T2[0],T1[0]
SUBC T2[1],T1[1]
SUBC T2[2],T1[2]
SUBC T2[3],T1[3]
SUBC T2[4],T1[4]
SUBC T2[5],T1[5]
SUBC T2[6],T1[6]
SUBC T2[7],T1[7]
SBC CW
```

Subroutine 5. Assembly subroutine for $t_1 - t_0 - t_2$ computation in 128-bit subtractive Karatsuba squaring.

```
SUB T1[0],T0[4]
SUBC T1[1],T0[5]
SUBC T1[2],T0[6]
SUBC T1[3],T0[7]
SUBC T1[4],T2[0]
SUBC T1[5],T2[1]
SUBC T1[6],T2[2]
SUBC T1[7],T2[3]
SUBC CW,T2[4]
ADC T2[5]
ADC T2[6]
ADC T2[7]
```

Subroutine 6. Subroutine for subtracting $t_1 - t_0 - t_2$ to finalize 128-bit subtractive Karatsuba squaring.

this optimization, we are able to reduce the number of word multiplications from 16 down to 10. We depict the computations performed in operand scanning multiplication and the optimizations performed in operand scanning squaring in Figure 5. Optimizing operand scanning multiplication for squaring accelerates the squaring computation considerably.

On MSP430F1611 and MSP430F2618, which have an onboard 16 × 16-bit hardware multiplier, 64-bit operand scanning *multiplication* takes 210 and 189 clock cycles, respectively. On the same microcontrollers, 64-bit operand scanning *squaring* takes 170 and 155 clock cycles, respectively. Hence, with 64-bit operand scanning squaring, we achieve 23% and 22% speedups over 64-bit operand scanning multiplication on MSP430F1611 and MSP430F2618, respectively. On the other target microcontroller, MSP430F5529, which has an onboard 32 × 32-bit hardware multiplier, 128-bit operand scanning *multiplication* takes 466 clock cycles. Whereas, on the same microcontroller, 128-bit operand scanning *squaring* takes 458 clock cycles. Hence, with 128-bit operand scanning squaring, we achieve around 2% speedup over 128-bit operand scanning multiplication on MSP430F5529.

4) COMBINED OPTIMIZATION THROUGH SUBTRACTIVE KARATSUBA-OFMAN SQUARING AND OPERAND-SCANNING SQUARING

The combined optimizations that come with operand scanning squaring and subtractive Karatsuba-Ofman squaring result in improved performance for integer squaring operations. On MSP430F1611 and MSP430F2618, which have an onboard 16 × 16-bit hardware multiplier, 128-bit subtractive Karatsuba-Ofman *multiplication* takes 921 and 839 clock cycles, respectively. Whereas, on the same microcontrollers, 128-bit subtractive Karatsuba-Ofman *squaring* takes 727 and 665 clock cycles, respectively. Hence, with 128-bit subtractive Karatsuba-Ofman squaring (with 64-bit operand scanning squaring at the base of recursion), we achieve 27% and 26% speedups over 128-bit subtractive Karatsuba-Ofman multiplication (with 64-bit operand scanning multiplication at the base of recursion), on MSP430F1611 and MSP430F2618, respectively.

On the other target microcontroller, MSP430F5529, which has an onboard 32 × 32-bit hardware multiplier, 256-bit subtractive Karatsuba-Ofman *multiplication* takes 1888 clock cycles. Whereas, on the same microcontroller, 256-bit subtractive Karatsuba-Ofman *squaring* takes 1742 clock cycles. Hence, with 256-bit subtractive Karatsuba-Ofman squaring (with 128-bit operand scanning squaring at the base of recursion), we achieve 8% speedup over 256-bit subtractive Karatsuba-Ofman multiplication (with 128-bit operand scanning multiplication at the base of recursion) on MSP430F5529.

The resulting performance of 2048-bit subtractive Karatsuba-Ofman squaring is dramatically better compared to the performance of 2048-bit subtractive Karatsuba-Ofman multiplication. For the microcontrollers MSP430F1611, MSP430F2618 and MSP430F5529, 2048-bit Karatsuba-Ofman *multiplication* takes 110733, 102167 and 73066 clock cycles, respectively. Whereas, on the same microcontrollers, 2048-bit subtractive Karatsuba-Ofman *squaring* takes 84720, 78479 and 63968 clock cycles, respectively. Hence, with 2048-bit subtractive Karatsuba-Ofman squaring (with operand scanning squaring at the base of recursion), we achieve 34%, 30% and 14% speedups over 2048-bit subtractive Karatsuba-Ofman multiplication (with operand scanning multiplication at the base of recursion) on MSP430F1611, MSP430F2618 and MSP430F5529, respectively.

B. SIDE-CHANNEL ATTACK COUNTERMEASURES

Side-channel attacks on cryptographic implementations vary in a wide range and several low-cost countermeasures have been proposed to mitigate them [47], [48], [49], [50], [51], [52]. The first attack that comes to mind is the SPA attack [33]. In the SPA attack, it is assumed that the secret key has an impact on the power consumption and the attacker is able to monitor it. Since the key pattern is visually identifiable in a power trace that is collected by the attacker, it can reveal the full length of secret key bits even from a single power trace [53]. The SPA attack can be applied to RSA decryption. In RSA decryption, exponentiation using the binary square and multiply method would reveal the secret exponent bits since a multiplication is performed, in addition to a squaring, only when the scanned secret key bit is 1. Since the difference between the square and the multiply operations can be distinguished even with the naked eye by looking at the power consumption trace of RSA decryption under an oscilloscope, the secret key would be revealed easily. Exponent blinding is believed to prevent the SPA attack on the binary square and multiply method; however, it is proved in [54] that this claim is not generally true for the chosen plaintext attack. The attack proposed in [54] can be prevented by combining exponent blinding (secret key blinding) with base blinding (ciphertext blinding). In [55], it is shown that an improved version of the binary square and multiply exponentiation algorithm, known as the multiply always exponentiation method, where the squaring operation is performed in the form of a multiplication operation, can be attacked successfully even though the secret key blinding and the ciphertext blinding countermeasures are applied together. In their work, they use unsupervised machine learning (ML) methods to differentiate whether a multiplication or a squaring operation is computed from the trace segments of power measurements. In our RSA implementations, we use the 4-bit sliding window method as an effective solution to prevent this attack. In the 4-bit sliding window method, the same computations (four squarings and one multiplication) are performed in every step of the exponentiation operation, regardless of the values of the exponent bits, to ensure indistinguishable power profiles. A similar approach that uses the 2-bit sliding window, as well as secret key blinding, is applied in [56]. Note that, in our work, we utilize all the mentioned side-channel attack

FIGURE 6. Fixed-time branch implementation (lines 4 – –6 in Algorithm 2) for the 1024-bit Montgomery multiplications in 2048-bit RSA decryption using the CRT.

countermeasures, namely the sliding window method (with the 4-bit window size), secret key blinding and ciphertext blinding. We explain the details of our implementation of these countermeasures in the rest of this section.

The running time characteristics of a cryptographic algorithm can be exploited by hackers to reveal secret information. Fixed-time implementations successfully mitigate timing attacks in exchange for some performance overhead. In order to alleviate the performance overhead of fixed-time implementations, the semi-fixed-time implementation, named as bucketing, is proposed [57]. However, in the semi-fixed-time implementation approach, there is a trade-off between security and performance which should be considered carefully. In our RSA implementations in this work, regardless of the performance trade-off, we implement all our arithmetic algorithms completely in fixed-time, as described in Section III-A. Another vulnerability of RSA against the SPA and timing attacks could be due to conditional branches, e.g. the conditional subtraction *if* $(T_1 \geq N)$ *then* $\{T_1 = T_1 - N\}$ in lines $4 - 6$ of Montgomery multiplication in Algorithm 2. Here, the conditional branch operation would leak information on secret data. In order to prevent this vulnerability, we eliminate the if statement and conduct the subtraction operation regardless of the if condition. We allocate a sign word, namely SW, for storing the sign of the result of the subtraction operation. Before executing the subtraction, we clear SW. Then we compute the subtraction $T_1 - N$ and write the result at an offset of D bytes away from the original location of T_1 in the memory where $D \geq 128$ for 1024-bit Montgomery multiplication as it takes place in 2048-bit RSA decryption using the CRT. Thus, if T_1 is stored at the address $Addr1$ in the memory, $T_1 - N$ will be stored at the address $Addr1+D$. After performing T_1-N, we subtract the resulting borrow bit from the sign word SW. Hence, SW is set to either 0×0000 or $0\times$FFFF, depending on whether the result of $T_1 - N$ is positive or negative, respectively. Thus, we have T_1 and $T_1 - N$, the result of Montgomery multiplication associated with both courses of action for the if statement, stored in the memory. All we need is to access the correct result in the memory and move it into the memory address for the output of Montgomery multiplication.

We compute the memory address for the location of the output of Montgomery multiplication by first computing the bitwise AND of D and SW which will result in the value D or 0 depending on whether SW is $0\times$FFFF or 0×0000, respectively. We then XOR this value with D and add the result as an offset to $Addr1$, the memory address of T_1. We read the result of the Montgomery multiplication computation from this memory address. Hence, the result Z of Montgomery multiplication is read either as T_1 from the address $Addr1$ or as $T_1 - N$ from the address $Addr1 + D$, and it is written to the memory address $Addr2$, as depicted in Figure 6.

The DPA attack is another possible side-channel attack on RSA which performs complex statistical analysis on decryption power traces to reveal the RSA decryption key [33], [60]. The DPA attack is feasible on a CRT implementation of RSA. However, it can be mitigated effectively if the multiplicative message blinding countermeasure is used [58]. Message blinding, also known as ciphertext randomization, is shown to be an effective countermeasure on modern PCs, against possible side-channel attacks that use trace measurements from the power supply, chassis potential or electromagnetic emanation [59]. For affordable DPA attack mitigation, we use the blinding method [22], [33]. We blind both the ciphertext and the secret key. For blinding the ciphertext, we multiply it with the random integer V_i before RSA decryption. After decryption, we recover the original plaintext by multiplying the result of RSA decryption with the second random integer V_f. For the method to work, we select the random pair (V_i, V_f) to satisfy the relationship

$$V_i = (V_f^{-1})^d \bmod N$$

where d is the decryption key and N is the RSA modulus. We precompute the random pair (V_i, V_f) and store it on the microcontroller. Therefore, no timing overhead is incurred due to its generation during RSA decryption. However, using the same (V_i, V_f) values repeatedly in different RSA decryptions may cause vulnerabilities. In order to efficiently overcome this issue, we alter V_i and V_f by updating them with their squares before each RSA decryption operation, as suggested in [33]. Note that for 2048-bit RSA, V_i and V_f are 2048-bit random integers. Hence, in our 2048-bit RSA implementation, message blinding is achieved with negligible timing overhead by doing only two 2048-bit modular squarings (for updating the random integers) and two 2048-bit modular multiplications (for blinding the ciphertext).

For further protection against DPA attacks, we also blind the secret exponent d, as suggested in previous

TABLE 1. Our timings for RSA with a 2048-bit key on the MSP430 microcontroller.

RSA 2048-bit Operation	Microcontroller	# Clock Cycles	Time (sec)
Encryption	MSP430F5529 @25 MHz	3,722,815	0.14
Encryption	MSP430F2618 @16 MHz	4,981,913	0.31
Encryption	MSP430F1611 @8 MHz	5,387,125	0.67
Decryption	MSP430F5529 @25 MHz	189,361,044	7.56
Decryption	MSP430F2618 @16 MHz	258,098,168	16.08
Decryption	MSP430F1611 @8 MHz	279,832,016	34.90

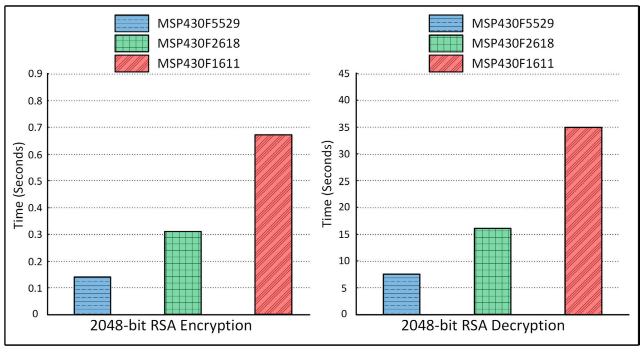


FIGURE 7. Our timings for 2048-bit RSA on the MSP430 microcontroller.

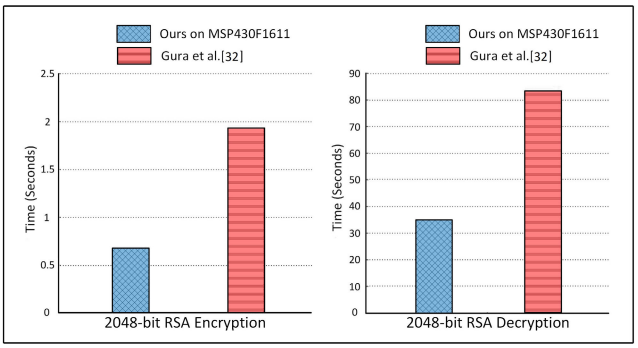


FIGURE 8. Comparison for 2048-bit RSA timings.

works [22], [33]. Exponent blinding is basically the randomization of the private decryption key d described as

$$d' = d + r \cdot \phi(N)$$

where r is a random integer and $\phi(N) = (p-1)\cdot(q-1)$. We apply exponent blinding to our RSA decryption implementation and thus our CRT exponents become $d'_p = d_p + r \cdot \phi(p)$ and $d'_q = d_q + r \cdot \phi(q)$ where $\phi(p) = p-1$, $\phi(q) = q-1$ and r is a random 32-bit integer as suggested in [22]. Exponent blinding increases the length of the exponent by 32 bits which results in a timing overhead of around 3% in decryption.

IV. PERFORMANCE EVALUATION

We implement 2048-bit RSA on three target MSP430 microcontrollers, namely MSP430F5529, MSP430F2618, and MSP430F1611. We develop our 2048-bit RSA implementation in C language. In addition, we write assembly subroutines for implementing core arithmetic operations. Our 2048-bit RSA encryption timings are all less than a second, i.e. 0.14 s, 0.31 s and 0.67 s on MSP430F5529, MSP430F2618 and MSP430F1611, respectively. Our 2048-bit RSA decryption timings are 7.56 s, 16.08 s and 34.90 s on MSP430F5529, MSP430F2618 and MSP430F1611, respectively. We use the same codes for the three MSP430 microcontrollers with slight adaptations to exploit the distinct features of the specific microcontrollers. For instance, while MSP430F5529 uses a 32 × 32 multiplier, MSP430F2618 and MSP430F1611 use a 16 × 16 multiplier. Additionally, MSP430F2611 and MSP430F5529 have a more advanced instruction set architecture, named MSP430X, which allows the memory write instruction to execute 1 clock cycle faster. Table 1 and Figure 7 present the timings of our RSA implementation on the three target MSP430 microcontrollers.

There are several existing implementations of RSA with a 1024-bit key on MSP430 or similar constrained microcontrollers [22], [23], [61], [62], [63]. However, to the best of our knowledge, there is only one other reported 2048-bit RSA implementation in the literature on a comparable constrained microcontroller, namely Gura et al.'s work on ATmega128 [32]. To make a fair evaluation, we compare with the work in [32] our 2048-bit RSA implementation on the MSP430F1611 microcontroller which has the same clock frequency and similar memory capacity. Our work on the low-end MSP430 microcontroller MSP430F1611 presents significantly better performance than the existing implementation in [32] as shown in Table 2 and Figure 8. Our 2048-bit RSA implementation utilizes the same techniques as those used in [32], namely small public exponent $e = 2^{16} - 1$, Montgomery modular multiplication and Chinese remainder theorem. However, we utilize additionally the subtractive Karatsuba-Ofman method for multiprecision integer multiplication. We implement subtractive Karatsuba-Ofman recursively for improved timing performance. We fully unroll the recursions in our subtractive Karatsuba-Ofman implementation by making inline assembly subroutine calls and thus reduce procedure call overheads. Moreover, we use loop unrolling in our codes as much as possible to reduce control flow overhead. Loop unrolling reduces the number of clock cycles needed for loop instructions, such as counter increment/decrement and conditional jump; however, it also increases the code size. Since our target microcontrollers have enough memory, we accelerate our implementations by unrolling the loops in our code as much as possible. For 512-bit subtractive Karatsuba-Ofman and all its underlying arithmetic operations, we fully unroll all the loop operations. For 1024-bit and larger subtractive Karatsuba-Ofman, we unroll loop operations as much as possible to reduce

TABLE 2. Timing comparisons for RSA implementations with a 2048-bit key on constrained microcontrollers.

RSA 2048-bit Operation	Microcontroller	Used Techniques	Memory Usage (Data & Code)	Time (s)
Encryption [Ours]	MSP430F1611 @8 MHz	$e = 2^{16} - 1$, Montgomery, Sub. Karatsuba-Ofman	1.5 kB & 8.2 kB	0.67
Encryption [32]	ATmega128 @8 MHz	$e = 2^{16} - 1$, Montgomery	1.3 kB & 2.8 kB	1.94
Decryption [Ours]	MSP430F1611 @8 MHz	CRT, Montgomery, Sub. Karatsuba-Ofman	3.8 kB & 13 kB	34.90
Decryption [32]	ATmega128 @8 MHz	CRT, Montgomery	1.8 kB & 7.7 kB	83.36

control flow overhead but keep some loops to save memory. Furthermore, unlike the existing implementation, our RSA implementation is equipped with the message and key blinding countermeasures to mitigate side-channel attacks. Our implementation has the drawback of using more memory compared to Gura et al.'s work due to the additional acceleration and side-channel protection methods used; however, it is significantly faster. While our implementation uses $\times 2.37$ and $\times 1.77$ more memory, it is $\times 2.90$ and $\times 2.39$ faster for encryption and decryption operations, respectively. We believe the resulting memory drawback is an acceptable trade-off for the achieved timing performance gain.

V. CONCLUSION

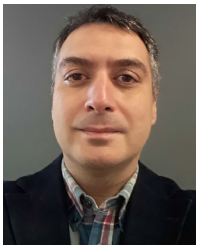
RSA is the oldest and the most commonly adopted public-key cryptographic algorithm that is utilized by the existing Internet infrastructure and related applications. We presented a practical and side-channel resistant implementation of 2048-bit RSA on a constrained microcontroller that is widely used in WSN nodes and IoT devices. Our fastest RSA implementation achieved 2048-bit encryption and decryption operations in 0.14 s and 7.56 s, respectively. Furthermore, our implementation on the low-end MSP430 microcontroller achieved 2048-bit RSA significantly faster ($\times 2.9$ and $\times 2.4$ for encryption and decryption) with respect to the existing implementation on the comparable ATmega128 microcontroller. We accomplished these performance figures by utilizing numerous acceleration methods, e.g. Montgomery multiplication, subtractive Karatsuba-Ofman, and CRT-based modular exponentiation. Furthermore, unlike the existing work, we implemented the necessary countermeasures to mitigate side-channel attacks, e.g. SPA and DPA, by utilizing the constant time implementation, sliding window, ciphertext blinding and secret key blinding methods.

REFERENCES

[1] B. D. Deebak and F. Al-Turjman, "A hybrid secure routing and monitoring mechanism in IoT-based wireless sensor networks," *Ad Hoc Netw.*, vol. 97, Feb. 2020, Art. no. 102022.
[2] R. Fotohi, S. F. Bari, and M. Yusefi, "Securing wireless sensor networks against denial-of-sleep attacks using RSA cryptography algorithm and interlock protocol," *Int. J. Commun. Syst.*, vol. 33, no. 4, 2020, Art. no. e4234.
[3] U. Banerjee, A. Wright, C. Juvekar, M. Waller, Arvind, and A. P. Chandrakasan, "An energy-efficient reconfigurable DTLS cryptographic engine for securing Internet-of-Things applications," *IEEE J. Solid-State Circuits*, vol. 54, no. 8, pp. 2339–2352, Aug. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8721457
[4] M. Mahbub, "A smart farming concept based on smart embedded electronics, Internet of Things and wireless sensor network," *Internet Things*, vol. 9, Mar. 2020, Art. no. 100161.
[5] Z. Zhang, S. Glaser, T. Watteyne, and S. Malek, "Long-term monitoring of the Sierra Nevada snowpack using wireless sensor networks," *IEEE Internet Things J.*, vol. 9, no. 18, pp. 17185–17193, Sep. 2022.
[6] D. E. Boubiche, A. K. Pathan, J. Lloret, H. Zhou, S. Hong, S. O. Amin, and M. A. Feki, "Advanced industrial wireless sensor networks and intelligent IoT," *IEEE Commun. Mag.*, vol. 56, no. 2, pp. 14–15, Feb. 2018.
[7] G. Pottie and L. Clare, "Wireless integrated network sensors: Toward low-cost and robust self-organizing security networks," *Proc. SPIE*, vol. 3577, pp. 86–95, Jan. 1999.
[8] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors," *Commun. ACM*, vol. 43, no. 5, pp. 51–58, 2000.
[9] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," *Comput. Netw.*, vol. 38, no. 4, pp. 393–422, 2002.
[10] S. Vollala, V. V. Varadhan, K. Geetha, and N. Ramasubramanian, "Design of RSA processor for concurrent cryptographic transformations," *Microelectron. J.*, vol. 63, pp. 112–122, May 2017.
[11] X.-J. Lin, L. Sun, and H. Qu, "An efficient RSA-based certificateless public key encryption scheme," *Discrete Appl. Math.*, vol. 241, pp. 39–47, May 2018.
[12] V. Dimitrov, L. Vigneri, and V. Attias, "Fast generation of RSA keys using smooth integers," *IEEE Trans. Comput.*, vol. 71, no. 7, pp. 1575–1585, Jul. 2022.
[13] G. Ganbaatar, D. Nyamdorj, G. Cichon, and T.-O. Ishdorj, "Implementation of RSA cryptographic algorithm using SN P systems based on HP/LP neurons," *J. Membrane Comput.*, vol. 3, pp. 22–34, Mar. 2021.
[14] H. Yu and Y. Kim, "New RSA encryption mechanism using one-time encryption keys and unpredictable bio-signal for wireless communication devices," *Electronics*, vol. 9, no. 2, p. 246, Feb. 2020.
[15] E. Ochoa-Jiménez, L. Rivera-Zamarripa, N. Cruz-Cortés, and F. Rodríguez-Henríquez, "Implementation of RSA signatures on GPU and CPU architectures," *IEEE Access*, vol. 8, pp. 9928–9941, 2020.
[16] R. Karim, L. S. Rumi, M. A. Islam, A. A. Kobita, T. Tabassum, and M. S. Hossen, "Digital signature authentication for a bank using asymmetric key cryptography algorithm and token based encryption," in *Evolutionary Computing and Mobile Sustainable Networks*, V. Suma, N. Bouhmala, and H. Wang, Eds. Singapore: Springer, 2021, pp. 853–859.
[17] O. F. A. Wahab, A. M. A. Khalaf, I. A. Hussein, and F. A. H. Hamed, "Hiding data using efficient combination of RSA cryptography, and compression steganography techniques," *IEEE Access*, vol. 9, pp. 31805–31815, 2021.
[18] K. Jiao, G. Ye, Y. Dong, X. Huang, and J. He, "Image encryption scheme based on a generalized Arnold map and RSA algorithm," *Secur. Commun. Netw.*, vol. 2020, Jun. 2020, Art. no. 9721675.
[19] K. G. M. Nag, Sharvari, D. V. Vaishnavi, S. Rajashree, and P. B. Honnavalli, "RSA implementation on sensor data in cold storage warehouse," in *Proc. IEEE Eurasia Conf. IoT, Commun. Eng. (ECICE)*, Oct. 2020, pp. 75–78. [Online]. Available: https://ieeexplore.ieee.org/document/9301979
[20] K. Pavani and P. Sriramya, "Enhancing public key cryptography using RSA, RSA-CRT and N-prime RSA with multiple keys," in *Proc. 3rd Int. Conf. Intell. Commun. Technol. Virtual Mobile Netw. (ICICV)*, 2021, pp. 1–6.
[21] Y. Fu, W. Wang, L. Meng, Q. Wang, Y. Zhao, and J. Lin, "VIRSA: Vectorized in-register RSA computation with memory disclosure resistance," in *Information and Communications Security*, D. Gao, Q. Li, X. Guan, and X. Liao, Eds. Cham, Switzerland: Springer, pp. 293–309, 2021.
[22] Z. Liu, J. Großschädl, and I. Kizhvatov, "Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers," in *Proc. Workshop Secur. Internet Things-SOCIOT*, vol. 10, 2010, pp. 1–10.
[23] E. Barker, *Recommendation for Key Management: Part 1—General*, Standard NIST SP 800-57, Part 1 Revision 5, 2020.
[24] E. Wenger and M. Werner, "Evaluating 16-bit processors for elliptic curve cryptography," in *Smart Card Research and Advanced Applications*. Berlin, Germany: Springer, 2011, pp. 166–181. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-27257-8_11#citeas

[25] U. Gulen and S. Baktir, "Elliptic curve cryptography for wireless sensor networks using the number theoretic transform," *Sensors*, vol. 20, no. 5, p. 1507, 2020.
[26] N. Koblitz and A. Menezes, "A riddle wrapped in an enigma," *IEEE Secur. Privacy*, vol. 14, no. 6, pp. 34–42, Nov./Dec. 2016.
[27] F. Karray, M. W. Jmal, A. Garcia-Ortiz, M. Abid, and A. M. Obeid, "A comprehensive survey on wireless sensor node hardware platforms," *Comput. Netw.*, vol. 144, pp. 89–110, Oct. 2018.
[28] WiSense Technologies. *WSN1120L Datasheet*. Accessed: Mar. 23, 2022. [Online]. Available: https://wisense.in/wp-content/uploads/2019/12/WSN1120L_Datasheet.pdf
[29] WiSense Technologies. *WSN1120CL Datasheet*. Accessed: Mar. 23, 2022. [Online]. Available: https://wisense.in/wp-content/uploads/2019/12/WSN1120CL_Datasheet.pdf
[30] WiSense Technologies. *WSN1101ANL Datasheet*. Accessed: Mar. 23, 2022. [Online]. Available: https://wisense.in/wp-content/uploads/2020/01/WSN1101ANL_datasheet.pdf
[31] WiSense Technologies. *WSN1101ACL Datasheet*. Accessed: Mar. 23, 2022. [Online]. Available: https://wisense.in/wp-content/uploads/2020/01/WSN1101ACL_Datasheet.pdf
[32] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," in *Cryptographic Hardware and Embedded Systems—CHES 2004*. Berlin, Germany: Springer, 2004, pp. 119–132.
[33] P. C. Kocher, "Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems," in *Advances in Cryptology—CRYPTO'96* (Lecture Notes in Computer Science), vol. 1109. Berlin, Germany: Springer, 1996, pp. 104–113. [Online]. Available: https://www.iacr.org/cryptodb/data/paper.php?pubkey=1469
[34] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.
[35] L. N. Childs, "RSA cryptography and prime numbers," in *Cryptology and Error Correction* (Springer Undergraduate Texts in Mathematics and Technology). Cham, Switzerland: Springer, 2019. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-15453-0_9, doi: 10.1007/978-3-030-15453-0_9.
[36] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
[37] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Sov. Phys. Doklady*, vol. 7, no. 7, pp. 595–596, 1963.
[38] M. Hutter and P. Schwabe, "Multiprecision multiplication on AVR revisited," *J. Cryptograph. Eng.*, vol. 5, no. 3, pp. 201–214, 2015.
[39] D. E. Knuth, *The Art of Computer Programmin: Seminumerical Algorithms*, vol. 2, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.
[40] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, 1st ed. Berlin, Germany: Springer, 2009.
[41] C. K. Koc, T. Acar, and B. S. Kaliski, "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.
[42] G. Hinterwälder, A. Moradi, M. Hutter, P. Schwabe, and C. Paar, "Full-size high-security ECC implementation on MSP430 microcontrollers," in *Progress in Cryptology—LATINCRYPT 2014* (Lecture Notes in Computer Science), vol. 8895, D. Aranha and A. Menezes, Eds. Cham, Switzerland: Springer, 2015. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-16295-9_2, doi: 10.1007/978-3-319-16295-9_2.
[43] *MSP430F552x, MSP430F551x Mixed-Signal Microcontrollers*, Texas Instrum., Dallas, TX, USA, 2015.
[44] *MSP430F261x, MSP430F241x Mixed-Signal Microcontroller*, Texas Instrum., Dallas, TX, USA, 2012.
[45] *MSP430F15x, MSP430F16x, MSP430F161x Mixed-Signal Microcontroller*, Texas Instrum., Dallas, TX, USA, 2011.
[46] *IDE Project Management and Building Guide for the Texas Instruments MSP430 Microcontroller Family*, IAR Syst., Uppsala, Sweden, 2021.
[47] M. J. Kannwischer, A. Genêt, D. Butin, J. Krämer, and J. Buchmann, "Differential power analysis of XMSS and SPHINCS," in *Constructive Side-Channel Analysis and Secure Design*, J. Fan and B. Gierlichs, Eds. Cham, Switzerland: Springer, 2018, pp. 168–188.
[48] M. Jurecek, J. Bucek, and R. Lórencz, "Side-channel attack on the A5/1 stream cipher," in *Proc. 22nd Euromicro Conf. Digit. Syst. Design (DSD)*, 2019, pp. 633–638.
[49] X. Duan, Q. Cui, S. Wang, H. Fang, and G. She, "Differential power analysis attack and efficient countermeasures on PRESENT," in *Proc. 8th IEEE Int. Conf. Commun. Softw. Netw. (ICCSN)*, Jun. 2016, pp. 8–12.
[50] J. Xu, A. Fan, M. Lu, and W. Shan, "Differential power analysis of 8-bit datapath AES for IoT applications," in *Proc. 17th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun./12th IEEE Int. Conf. Big Data Sci. Eng. (TrustCom/BigDataSE)*, Aug. 2018, pp. 1470–1473.
[51] M. Petrvalsky, T. Richmond, M. Drutarovsky, P. Cayrel, and V. Fischer, "Differential power analysis attack on the secure bit permutation in the McEliece cryptosystem," in *Proc. 26th Int. Conf. Radioelektronika (RADIOELEKTRONIKA)*, 2016, pp. 132–137.
[52] S.-P. Gao, Y. Guo, Z. T. Aung, and Y.-X. Guo, "Analysis of information leakage from MCU using neural network," in *Proc. 12th Int. Workshop Electromagn. Compat. Integr. Circuits (EMC Compo)*, 2019, pp. 171–173.
[53] T. Popp, S. Mangard, and E. Oswald, "Power analysis attacks and countermeasures," *IEEE Design Test Comput.*, vol. 24, no. 6, pp. 535–543, Nov./Dec. 2007.
[54] W. Schindler, "Exclusive exponent blinding may not suffice to prevent timing attacks on RSA," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2015, pp. 229–247.
[55] A. Kulow, T. Schamberger, L. Tebelmann, and G. Sigl, "Finding the needle in the haystack: Metrics for best trace selection in unsupervised side-channel attacks on blinded RSA," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 3254–3268, 2021.
[56] A. Aljuffri, C. Reinbrecht, S. Hamdioui, and M. Taouil, "Multi-bit blinding: A countermeasure for RSA against side channel attacks," in *Proc. IEEE 39th VLSI Test Symp. (VTS)*, Apr. 2021, pp. 1–6.
[57] B. Köpf and M. Dürmuth, "A provably secure and efficient countermeasure against timing attacks," in *Proc. 22nd IEEE Comput. Secur. Found. Symp.*, Jul. 2009, pp. 324–335.
[58] B. D. Boer, K. Lemke, and G. Wicke, "A DPA attack against the modular reduction within a CRT implementation of RSA," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2002, pp. 228–243.
[59] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs," *J. Cryptograph. Eng.*, vol. 5, pp. 95–112, May 2015.
[60] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *J. Cryptograph. Eng.*, vol. 1, pp. 5–27, Mar. 2011.
[61] U. Gulen, A. Alkhodary, and S. Baktir, "Implementing RSA for wireless sensor nodes," *Sensors*, vol. 19, no. 13, p. 2864, 2019.
[62] L. Qiu, Z. Liu, G. C. C. F. Pereira, and H. Seo, "Implementing RSA for sensor nodes in smart cities," *Pers. Ubiquitous Comput.*, vol. 21, no. 5, pp. 807–813, Oct. 2017.
[63] H. Wang and Q. Li, "Efficient implementation of public key cryptosystems on mote sensors (short paper)," in *Information and Communications Security* (Lecture Notes in Computer Science), vol. 4307, P. Ning, S. Qing, and N. Li, Eds. Berlin, Germany: Springer, 2006. [Online]. Available: https://link.springer.com/chapter/10.1007/11935308_37, doi: 10.1007/11935308_37.

UTKU GULEN received the B.Sc. degree in electronics and communication engineering from Yildiz Technical University, Istanbul, Turkey, in 2012, and the M.Sc. and Ph.D. degrees in computer engineering from Bahcesehir University, Istanbul, in 2014 and 2022, respectively. His research interests include applied cryptography, network and computer security, public-key cryptography on IoT devices, and embedded systems.



SELCUK BAKTIR (Member, IEEE) received the B.Sc. degree in electrical engineering from Bilkent University, Ankara, Turkey, in 2001, and the M.Sc. and Ph.D. degrees in electrical and computer engineering from the Worcester Polytechnic Institute, MA, USA, in 2003 and 2008, respectively.

He worked for companies, including IBM T. J. Watson Research Center and Intel Corporation. In 2013, he founded the M.Sc. Program in Cybersecurity with Bahcesehir University, Istanbul, Turkey. His research interests include applied cryptography and computer security. He received the IBM Research Pat Goldberg Memorial Best Paper Award, in 2007, the European Union FP7 Marie Curie IRG Award, in 2010, and the TUBITAK Career Award, in 2016.