PUBLIC-KEY CRYPTOGRAPHY ON CONSTRAINED EMBEDDED DEVICES

Utku GÜLEN

JUNE 2022

PUBLIC-KEY CRYPTOGRAPHY ON CONSTRAINED EMBEDDED DEVICES

A PhD THESIS SUBMITTED TO THE

GRADUATE SCHOOL

OF

BAHÇEŞEHİR UNIVERSITY

BY

UTKU GÜLEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE PhD DEGREE OF COMPUTER ENGINEERING

IN THE DEPARTMENT OF BAHCESEHIR UNIVERSITY

JUNE 2022

T.C.

BAHÇEŞEHİR UNIVERSITY

GRADUATE SCHOOL

…/…/…

PhD THESIS APPROVAL FORM

Name Surname	UTKU GÜLEN
Student Number	1407292
Program Name	COMPUTER ENGINEERING (ENGLISH - DOCTORATE)
Title of Thesis	PUBLIC-KEY CRYPTOGRAPHY ON CONSTRAINED EMBEDDED DEVICES
Thesis Defense Date	

It has been approved by the Graduate School that this thesis has fulfilled the necessary conditions as a PhD thesis.

Prof. Dr. Ahmet ÖNCÜ

Director of the Graduate School

This Thesis has been read by us, it has been deemed sufficient and accepted as a PhD thesis in terms of quality and content.

PhD Thesis Defense Jury		
Thesis Defense Jury	**Title - Name / Surname**	**Signature**
Thesis Advisor	ASST. PROF. DR. SELÇUK BAKTIR	
Member of Thesis Monitoring Committee	PROF. DR. SIDDIKA BERNA ÖRS YALÇIN	
Member of Thesis Monitoring Committee	ASST. PROF. DR. TARKAN AYDIN	
Member	ASSOC. PROF. DR. ALPTEKIN KÜPÇÜ	
Member	ASST. PROF. DR. ECE GELAL SOYAK	

I hereby declare that all information in this document has been obtained and presentedin accordance with academic rules and ethical conduct. I also declare that, as requiredby these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: UTKU GÜLEN

Signature:

ABSTRACT

PUBLIC-KEY CRYPTOGRAPHY ON CONSTRAINED EMBEDDED DEVICES

GÜLEN, Utku
Computer Engineering (English) PhD Program

Supervisor: Asst. Prof. Dr. Selçuk BAKTIR

JUNE 2022, 68 pages

Constrained embedded devices are widely deployed in many areas including wireless sensor network and Internet of things. Since such applications may contain sensitive data and may serve to critical purposes, securing the communication between the wireless sensor nodes or Internet of things devices is vital. Applying only symmetric key cryptography remains with a problem when it comes to distribution of the private keys. Thus, it is necessary to use a public-key cryptography for sharing private keys effectively. With its relatively smaller key size, elliptic curve cryptography is usually the first choice in literature. However, acceleration methods are still required since the computations are heavy for the constrained devices such as microcontrollers. For this reason, we implemented elliptic curve cryptography on a constrained microcontroller, MSP430, utilizing the number theoretic transform based computations, optimal extension field and Edwards curve. We presented our applied methods and the performance result including the comparison with the literature in the first part of the thesis. Another public-key cryptography, Rivest Shamir Adleman (RSA), has a larger key size which is a drawback for the resource constrained microcontrollers. However, it allows fast digital signature verification and can be desirable since it's widely used in the existing communication infrastructure. In the second part of the thesis, it is showed that, in spite of its long key size, RSA is applicable for constrained devices when optimized arithmetic, low-level coding and some acceleration algorithms are used. RSA implementations that are presented in the thesis have the fastest reported timings compared to the studies in the literature which use similar constrained microcontrollers.

Keywords: Wireless Sensor Networks, Internet of Things, Public-Key Cryptography, Elliptic Curve Cryptography, Rivest Shamir Adleman

ÖZET

KISITLI KAYNAKLI GÖMÜLÜ CİHAZLAR ÜZERİNDE AÇIK ANAHTARLI ŞİFRELEME

GÜLEN, Utku

Bilgisayar Mühendisliği (İngilizce) Doktora Programı

Danışman: Dr. Öğretim Üyesi Selçuk BAKTIR

HAZİRAN 2022, 68 sayfa

Kısıtlı kaynaklı gömülü cihazlar, kablosuz sensör ağları ve nesnelerin İnterneti de dahil olmak üzere bir çok uygulamada konuşlandırılmaktadır. Bu uygulamalar hassas verilere sahip olabileceğinden ve kritik amaçlara hizmet edebileceğinden, kablosuz sensör düğümleri veya nesnelerin İnterneti cihazları arasındaki haberleşmenin güvenliğinin sağlanması önemlidir. Sadece simetrik şifreleme uygulanması, gizli anahtarların dağıtılması problemini çözümsüz bırakır. Bu nedenle, gizli anahtar dağıtımını etkin bir şekilde yapabilmek için, açık-anahtarlı şifreleme kullanılmalıdır. Görece daha kısa anahtarı olduğundan, eliptik eğri şifreleme genellikle literatüredeki ilk seçimdir. Fakat, mikrodenetleyiciler gibi kısıtlı kaynaklı cihazlar için hesaplamalar ağır olduğundan, hızlandırma yöntemleri hala gerekmektedir. Bu nedenle, kısıtlı kaynaklı bir mikrodenetleyici olan MSP430 için, eliptik eğri şifrelemeyi sayılar teorisi dönüşümü tabanlı işlemler, en uygun ilave alan ve Edwards eğrileri kullanarak gerçekleştirdik. Uyguladığımız yöntemleri ve literatürdeki diğer çalışmarla beraber olan performans sonuçlarını tezin ilk kısmında sunmaktayız. Bir diğer açık anahtarlı şifreleme olan Rivest Shamir Adleman (RSA) daha büyük boyutlu bir anahtara sahiptir ki bu kısıtlı kaynaklı mikrodenetleyiciler için bir dezavantajdır. Yine de hızlı sayısal imza doğrulama ve hali hazırdaki haberleşme yapılarında kullanıldığı için RSA tercih edilebilir. Tezin ikinci kısmında, büyük boyutlu anahtarı olmasına rağmen, en etkin hesaplama, düşük seviye kodlama ve bazı hızlandırma yöntemleri kullanarak RSA kullanılabilirliği gösterilmiştir. Bu tezde sunulan RSA gerçekleştirmeleri, literatürde benzer mikrodenetleyiciler kullanan çalışmalara kıyasla en hızlı zamanlamalara sahiptir.

Anahtar kelimeler: Kablosuz Sensör Ağları, Nesnelerin İnterneti, Açık-Anahtarlı Şifreleme, Eliptik Eğri Şifreleme, Kısıtlı Kaynaklı Gömülü Cihazlar

DEDICATION

To my family;
to my beloved wife Ana Maria,
to my parents Güler and Süleyman,
to my brother Umur.

ACKNOWLEDGMENTS

Firstly, I would like to express my deepest and sincerest gratitude for my advisor, Prof. Selçuk Baktır, who has inspired and guided me throughout my thesis studies. I would like to thank him, for sharing his wisdom with me and for his endless support. It has been an honor and a pleasure to study under his mentorship.

I am honored to have the thesis committee with very valuable professors. I would like to thank my thesis committee monitoring members, Prof. Sıddıka Berna Örs Yalçın and Prof. Tarkan Aydın for their valuable time and constructive suggestions which make my thesis significantly improved and richer. I would like to thank Prof. Alptekin Küpçü, for his insightful suggestions and beneficial comments in my dissertation. My sincere thanks to Prof. Ece Gelal Soyak, for her valuable comments in my dissertation and for her great help during the thesis processes.

I would like to thank all my teachers and professors from Bahcesehir University, Yildiz Technical University and earlier, who taught me and inspired me to pursue a doctoral degree.

I had great times working as a researcher and teaching assistant in Bahcesehir University. I would like to thank all my friends, my former colleagues, and the professors.

I am very pleased to work with very knowledgeable researchers and experts in Ericsson Research. The experience in Ericsson Research Turkey site, has broaden my perspective in security research greatly. I would like to thank all my colleagues in security team in Ericsson Research Turkey, my managers Dr. Emrah Tomur and Dr. Henrik Almeida for their support and guidance.

The happiness and success are more delightful when there are people who share and celebrate with you. I would like to thank all my friends for being with me during my PhD.

The most importantly, I would like to thank my dear family. I am very lucky to have them.

My deepest gratitude to my first teacher and my dear mother Güler Gülen, to my father Süleyman Gülen and to my brother Umur Gülen. They have always believed in me.

I would especially like to thank to my precious wife Ana Maria Noriega. She was with me, during the toughest times. We are and will be a good team, forever.

İstanbul, 2022 Utku Gülen

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
DFT	Discrete Fourier Transform
ECC	Elliptic Curve Cryptography
FFT	Fast Fourier Transform
GF	Galois Field
IAR	Ingenjörsfirman Anders Rundgren
IFFT	Inverse Fast Fourier Transform
IOT	Internet of Things
NAF	Non-adjacent Form
NTT	Number Theoretic Transform
NIST	National Institute of Standards and Technology
OEF	Optimal Extension Field
PM	Pairwise Multiplication
RISC	Reduced Instruction Set Computing
RSA	Rivest-Shamir-Adleman
WSN	Wireless Sensor Network

Chapter 1
Introduction

A wireless sensor network (WSN) consists of a large number of sensor nodes which are usually controlled by a constrained microcontroller, e.g., 16-bit MSP430, 8-bit ATmega, etc. Applications of WSNs vary widely from environmental practices to health, and from industrial control applications to military purposes (Akyildiz et al., 2007; Chong & Kumar, 2003; Pottie & Kaiser, 2000; Yick et al., 2008). WSNs also find applications in areas such as multimedia networks (Akyildiz et al., 2007) and smart grid networks (Gungor et al., 2010). WSN applications in diverse fields rely on underlying algorithms, protocols and standards (Baronti et al., 2007). Moreover, many WSN applications need data security and confidentiality since sensitive information is stored, processed or transferred by sensor nodes (Chen et al., 2009; Y. Wang et al., 2006). As the number and diversity of WSN applications grow, so do the variety of their security and privacy issues (Li et al., 2010; Ozdemir & Xiao, 2009; Roman et al., 2007). Therefore, it is a necessity to implement cryptographic schemes efficiently on constrained microcontrollers used in sensor nodes (He et al., 2015; Perrig et al., 2002; Y. Yu et al., 2012). Microcontrollers on WSN nodes are typically limited by their memory capacity and CPU speed. Besides, energy efficiency is another important constraint since WSN nodes are either battery powered or they environmentally harvest their energy (Chandrakasan et al., 1999; Ye et al., 2002; Zhou et al., 2008). All these constraints make implementing cryptography, more particularly complex public key cryptographic operations, on WSN nodes a major challenge. Lightweight cryptography also comes with the issues for the designers, while balancing security and performance trade-offs (Eisenbarth et al., 2007). And thus, it is considered less secure compared to conventional cryptography (Buchanan et al., 2017; Mouha, 2015).

For providing security to WSN nodes, symmetric key cryptography may seems to be a feasible solution at first glance. However, symmetric key cryptography alone would not be a remedy for providing security to WSN nodes. WSN nodes are typically placed far apart from each other and the distribution of shared symmetric keys is a challenge. For distributing the shared symmetric keys efficiently, public key cryptography (PKC) is needed (Diffie et al., 1976; Elgamal, 1985; Ingemarsson et al., 1982; MacKenzie et al., 2010). Furthermore, PKC makes it possible to electronically sign digital messages (Rivest et al., 1978). In WSNs, malicious nodes can inject false or counterfeit messages into the network. Hence, implementation of an authentication mechanism would be necessary to prevent these messages. Digital signatures, provided by PKC, can be used to authenticate messages exchanged between sensor nodes. Although the computational complexity of PKC is considered a drawback for WSN nodes with constrained microcontrollers, previous works show that PKC can be a viable option (Düll et al., 2015; Gouvêa & López, 2009; Gülen & Baktir, 2014; Gulen & Baktir, 2016; Z. Liu et al., 2010; Szczechowiak et al., 2008). The two most popular PKC schemes are elliptic curve cryptography (ECC) (Koblitz, 1987) and the Rivest Shamir Addleman (RSA) cryptosystem (Miller, 1986; Rivest et al., 1978). Many of the previous PKC implementations for WSNs in the literature prefer ECC due to its shorter key size (A. Liu & Ning, 2008; H. Wang et al., 2006; Wenger & Werner, 2011). ECC requires at least a 160-bit key to be considered secure. In RSA, on the other hand, the same level of security can be achieved with a 1024-bit key. However, despite its larger key size, RSA is more widely used particularly by general purpose computers in Internet applications.

1.1 Contribution in the Thesis

In our study (Gulen & Baktir, 2016), for the first time in literature, we showed that frequency domain arithmetic can be practically applied to ECC implementations on low-power microcontrollers without hardware multiplier support. We realized, for the first time, elliptic curve scalar point multiplication with a fixed point, as well as with a random point, in the frequency domain without using hardware multiplier support. Our implementations exhibit comparable timing performance to existing ECC implementations which use hardware multiplier support. Since our implementations don’t use hardware multiplier support, they are expected to be more power-efficient while having acceptable timing performance.

In our study (Gulen & Baktir, 2020), we present a novel realization of ECC which uses Edwards curves for point arithmetic and the Number Theoretic Transform (NTT) for the underlying finite field multiplication and squaring operations. To the best of our knowledge, our work presents the first realization of ECC using the Fast Fourier Transform (FFT) to speed up NTT computations. Our implementation achieves similar or faster timings for ECC scalar point multiplication compared to existing implementations in the literature and proves that NTT-based arithmetic is feasible for ECC implementations on constrained devices such as WSN nodes.

In our study (Gülen & Baktir, 2022), we utilize the FFT over a finite field to implement ECC on a constrained microcontroller without using hardware multiplier support for the first time in the literature. Over $GF((2^{13}-1)^{13})$, we achieve ECC point multiplication of random points in 1.74 s which is 13% faster than the existing work in (Gulen & Baktir, 2016). Furthermore, we achieve ECC point multiplication of fixed points in 0.89 s which is 10% faster than the existing work. Our proposed implementation with the FFT achieves ECC random and fixed point multiplication consuming 29.81 mWs and 15.27 mWs which are 15% and 12% less than the energy consumed by the existing implementation. We show that, in terms of both timing performance and energy consumption, FFT based multiplication would result in better performance for ECC than frequency domain Montgomery multiplication. With our proof-of-concept implementation, we show that on an extremely constrained platform that does not use a hardware multiplier, ECC can be performed efficiently when the FFT is used. Power savings gained through our proposed implementation would be significant for battery powered WSN nodes whose lifetime is limited by their stored energy, and more particularly for energy harvesting WSN nodes which harness energy from the environment and may have more strict power constraints.

In our study (Gulen et al., 2019), we use the subtractive Karatsuba-Ofman, Montgomery multiplication, Chinese remainder theorem (CRT), and operand scanning algorithms together for the first time in the literature, and implement RSA using these. Our 1024-bit RSA encryption and decryption implementations on the MSP430 microcontroller have the fastest timings in the literature. We show that faster RSA timings are feasible on WSN nodes and the RSA cryptosystem may be a preferable PKC option for WSNs.

In our recent study, we present efficient RSA implementations on the MSP430 family of constrained microcontrollers using a 2048-bit key as recommended by the NIST (Barker, 2020). We combine the acceleration techniques sliding window method, 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 (Gura et al., 2004), and achieves RSA encryption and decryption operations more than twice faster. Unlike the existing work in (Gura et al., 2004), our 2048-bit RSA implementation includes the necessary countermeasures to prevent vulnerabilities that may arise from implementation attacks such as simple power analysis (SPA) and differential power analysis (DPA) (P. C. Kocher, 1996). We show that strong RSA cryptography with a 2048-bit key is feasible on constrained microcontrollers used in WSN and IoT applications.

1.2 Organization

The thesis continues with Chapter 2, ECC implementations on constrained MSP430 microcontrollers which are used widely in WSN and IoT. Chapter 2 composed of sections which explain the background of the methods that we utilized, namely DFT based Montgomery multiplication, FFT based multiplication and Edwards curves. Then the optimization methods for the ECC are explained in detail, namely field arithmetic optimizations and point arithmetic optimizations for the FFT based multiplication. Then the chapter continues with comprehensive performance evaluation and comparison of our ECC with related works in the literature. The last part concludes Chapter 2 with our findings.

In Chapter 3, we present the techniques namely, sliding window method, CRT, Montgomery modular multiplication and subtractive Karatsuba-Ofman multiplication which we utilized in order to accelerate our 1024-bit and 2048-bit RSA operations. After introducing the applied methods, in the second part of the Chapter 3, we explain how we applied further low-level optimizations and how we realized our implementation to be execute in fixed time. Then we explain our side-channel attack countermeasures on RSA, i.e. SPA and DPA. Before concluding the Chapter 3, we present our implementation timings and compare with the existing works in the literature.

Chapter 4 concludes the studies in this thesis with remarks and discusses possible future works, finally.

Chapter 2
Elliptic Curve Cryptograph For Wireless Sensor Network Nodes Using Number Theoretic Transform

ECC (Koblitz, 1987; Miller, 1986) is a commonly used public-key cryptosystem and considered a viable remedy for distributing the secret keys in WSNs (Gouvêa et al., 2012; Gülen & Baktir, 2014; Gulen & Baktir, 2016; Z. Liu et al., 2010; Szczechowiak et al., 2008). The efficiency of ECC depends on the speed of the performed arithmetic. While projective coordinates are typically used to avoid expensive inversion, multiplication still needs to be performed. A word multiplication instruction typically takes much longer to execute than a word addition instruction on constrained microcontrollers. On some microcontrollers, for power efficiency and cost reasons, a multiplication circuitry does not even exist and word multiplications are implemented with shift and add instructions. For instance, the MSP430F1232, MSP430F2274 and MSP430G2955 versions of the MSP430 microcontroller are some of the several available microcontrollers which do not have a hardware multiplier (Texas Instruments, 2004). Note that, among these microcontrollers, the MSP430G2955 microcontroller is used in WiSense sensor nodes, namely the sensor nodes WSN1120L, WSN1120CL, WSN1101ANL and WSN1101ACL (WiSense Technologies, 2019). Moreover, Texas Instruments' (TI) development tool for wireless sensor applications, named as the TI eZ430- RF2500 wireless module, is also equipped with MSP430F2274 (Texas Instruments, 2015). Using a simple power efficient microcontroller is particularly important for wireless sensor network nodes which are spread around in the field and harvest their energy from the environment (Adu-Manu et al., 2018) . For energy-harvesting wireless sensor nodes, it is a concern whether the sensor node is able to perform a power-hungry cryptographic algorithm within the limitations of the harnessed power obtained through solar energy, mechanical vibration, electromagnetic radiation, etc. In (Gulen & Baktir, 2016), a competent ECC implementation on the constrained MSP430 microcontroller is proposed over the optimal extension field (OEF) $GF\ ((2^{13} - 1)^{13})$ (Bailey & Paar, 2001). The implementation uses the number theoretic transform and Edwards curve point arithmetic. For power efficiency, no hardware multiplier is used, and arithmetic operations are carried out in the frequency domain. In their implementation, the number theoretic transform is utilized to initially carry elliptic curve point coordinates to the frequency domain. All arithmetic operations required in ECC point multiplication are then conducted in the frequency domain. Montgomery multiplication is used for performing multiplication in the frequency domain (Baktir & Sunar, 2006; Baktır et al., 2007; Montgomery, 1985).

2.1 Background

ECC is performed over a finite field. Hence, picking an efficient finite field representation and using efficient arithmetic algorithms over the selected finite field significantly effects the performance of ECC. We implement ECC over an OEF. The OEF representation is an efficient finite field representation that is proposed for implementing ECC on constrained devices (Bailey & Paar, 2001). The OEF representation constructs the finite field $GF\ (p^m)$ by choosing p as a pseudo-Mersenne prime, such that GF (p) elements fit in a single processor register, and by using an irreducible binomial of the form $x^m - w$ where w is a small integer. The special forms of p and w facilitate efficient coefficient arithmetic and modular reduction. By fitting GF (p) elements in a single register word, only a single instruction cycle is spent to execute microcontroller instructions over GF (p) elements. When the extension degree m is selected as a prime number, ECC over GF (p^m) is considered secure (Koblitz et al., 2004). In our works, we implement ECC over GF (p^m) where $m = 13$ is prime and $p = 2^{13} - 1$ is a Mersenne prime. Selecting p as a Mersenne prime allows for very efficient reduction modulo p which is an operation commonly performed in GF (p^m) arithmetic. The finite field GF $((2^{13} - 1)^{13})$ is used in (Baktır et al., 2007; Mentens et al., 2015) for constrained hardware implementations of ECC and proved efficient.

On an elliptic curve defined over GF (p^m), the coordinates of a curve point are GF (p^m) elements and represented as degree $m - 1$ polynomials whose coefficients are in GF (p) (Baylis, 1988; McEliece, 1993). In ECC a large number of finite field divisions, multiplications, subtractions and additions are carried out. The subtraction/addition of $a(x)$ with $b(x)$ in GF (p^m) is achieved easily through pairwise modular word additions/subtractions of their polynomial coefficients, as given below:

$$a(x) \pm b(x) = \sum_{j=0}^{m-1} (a_j \pm b_j)\, x^j \bmod p \tag{1}$$

Whereas, multiplication of *GF* (p^m) elements is significantly more complex and necessitates a quadratic number of coefficient multiplications modulo p and a final modular reduction with the field polynomial, given as follows:

$$r'(x) = a(x).b(x) = \sum_{j=0}^{2m-1} (r_j' x^j), \tag{2}$$
$$r(x) = r'(x) \bmod p(x)$$

where $p(x)$ can be selected as $x^m - 2$ to make modular reduction simple. In this work, we use $p(x) = x^m - 2$ to construct the finite field *GF* ($(2^{13} - 1)^{13}$). Polynomial multiplication requires computing a quadratic number of expensive modular coefficient multiplications. The convolution theorem states that time domain polynomial multiplication produces the same result as frequency domain pairwise coefficient multiplications. Hence, the number of performed coefficient multiplications is reduced dramatically if frequency domain arithmetic is used for polynomial multiplication. DFT, or its optimized form FFT, can be used for carrying *GF* (p^m) elements into frequency domain.

2.1.1 Discrete Fourier transform based Montgomery multiplication. Algorithm 1 was proposed for achieving *GF* (p^m) multiplication using discrete Fourier transform based Montgomery multiplication (Baktir & Sunar, 2006; Baktır & Sunar, 2008b, 2008a). The algorithm was utilized in ECC implementations for constrained wireless sensor nodes (Baktır et al., 2007; Gulen & Baktir, 2016). The algorithm takes as inputs ($\bar{A}$) and ($\bar{B}$), which are the frequency domain series for $\bar{a}(x) = a(x)\, x^{m-1}$, $\bar{b}(x) = b(x)\, x^{m-1} \in GF\,(p^m)$ and produces their Montgomery product. Note that $\bar{a}(x)$ and $\bar{b}(x)$ are the Montgomery forms of $a(x)$ and $b(x)$. The output of the algorithm is denoted with $\bar{R}$ and represents the Montgomery product of $\bar{a}(x)$ and $\bar{b}(x)$, i.e. $\bar{a}(x)\, \bar{b}(x)\, x^{-(m-1)} \bmod p(x)$. Note that $\bar{R} = \bar{a}(x)\, \bar{b}(x)\, x^{-(m-1)} \bmod p(x)$ is equal to $a(x)\, b(x)\, x^{(m-1)} \bmod p(x)$ which is the Montgomery form of the product of $a(x)$ and $b(x)$ in *GF* (p^m). Using $p = 2^m - 1$, a Mersenne prime, results in more efficient DFT computations (Rader, 1972).

In Algorithm 1 (Gulen & Baktir, 2016), a linear number of word products are computed. Whereas, the number of performed bitwise rotations, subtractions and additions are quadratic. Since multiplication is more complex compared to other arithmetic operations on a constrained microcontroller, Algorithm 1 is desirable.

Algorithm 1: Discrete Fourier transform based Montgomery multiplication over $GF(p^m)$, with $p = 2^m - 1$ and $p(x) = x^m - 2$

```
Input: $(\bar{A})$ and $(\bar{B})$ are the frequency domain series for $\bar{a}(x) = a(x)x^{m-1}$ and
       $\bar{b}(x) = b(x)x^{m-1}$ in $GF(p^m)$
Output: $\bar{R} = \bar{a}(x)\bar{b}(x)x^{-(m-1)} \bmod x^m - 2$
1  for $j \leftarrow 0$ to $2m - 1$ do
2      $\bar{R}_j \leftarrow \bar{A}_j \bar{B}_j \bmod p$
3  end
4  for $i \leftarrow 0$ to $m - 2$ do
5      $T \leftarrow \bar{R}_0$
6      for $k \leftarrow 1$ to $2m - 1$ do
7          $T \leftarrow T + \bar{R}_k \bmod p$
8      end
9      $T \leftarrow -T/2m \bmod p$
10     $T_e \leftarrow T/2 \bmod p$
11     $T_o \leftarrow T + T_e \bmod p$
12     for $j \leftarrow 0$ to $m - 1$ do
13         $\bar{R}_{2j} \leftarrow (\bar{R}_{2j} + T_e)/2^{2j} \bmod p$
14         $\bar{R}_{2j+1} \leftarrow -(\bar{R}_{2j+1} + T_o)/2^{2j+1} \bmod p$
15     end
16 end
17 Return $\bar{R}$
```

2.1.2 Fast Fourier transform based multiplication.

In a recent work, ECC is implemented using a different DFT based approach to realize $GF(p^m)$ multiplications and squarings (Gulen & Baktir, 2020). In (Gulen & Baktir, 2020), the FFT (Baktır & Sunar, 2008a; Cooley & Tukey, 1965; Pollard, 1971) is used to transform $GF(p^m)$

elements into the frequency domain. Once the frequency domain representations for *GF* (p^m) elements are obtained, their polynomial multiplication is computed simply by pairwise multiplying their frequency domain coefficients. Utilizing the inverse fast Fourier transform (IFFT) algorithm, the resulting product is carried to the time domain. In Algorithm 2 (Gülen & Baktir, 2022), we present this approach which is composed of the three stages: FFT, pairwise multiplication (PM) and IFFT. With this work, similar to (Gulen & Baktir, 2020), we use FFT based multiplication to implement ECC. However, unlike in (Gulen & Baktir, 2020), we implement FFT based ECC without using hardware multiplier support to show that ECC can be achieved practically on an extremely constrained microcontroller without even a hardware multiplier. Furthermore, we obtain the energy consumption profile of our ECC implementation and show that it is more energy efficient than the existing implementation in (Gulen & Baktir, 2016) which also does not use hardware multiplier support.

```
Algorithm 2: Fast Fourier transform based multiplication
  Input: $a(x)$ and $b(x)$ in $GF(p^m)$
  Output: $r(x) = a(x)b(x) \bmod p(x)$
1 $(A) \longleftarrow FFT(a(x))$
2 $(B) \longleftarrow FFT(b(x))$
3 $(R) \longleftarrow PM((A),(B))$
4 $r(x) \longleftarrow IFFT((R))$
5 Return $r(x)$
```

2.1.2.1 Conversion of operands into the frequency domain. The finite field elements $a(x)$, $b(x) \in GF\ ((2^{13}-1)^{13})$ are carried to the frequency domain with the DFT as

$$A_i = \sum_{j=0}^{25} a_j e^{ji} \bmod p\,, 0 \le i \le 25 \tag{3}$$

and

$$B_i = \sum_{j=0}^{25} b_j e^{ji} \bmod p, 0 \le i \le 25 \tag{4}$$

where e is a 26th primitive root of unity. Algorithm 3 (Gulen & Baktir, 2020; Gülen & Baktir, 2022) performs the above DFT computations over $GF((2^{13}-1)^{13})$ efficiently by using the FFT. It is designed such that the seven general purpose registers on MSP430 are used heavily to store intermediary results so that the least number of time-consuming memory read/write instructions are executed. The algorithm uses the binomial $x^{13}-2$ as field generating polynomial and $e = -2$ as the 26th primitive root of unity. For the selected finite field $GF((2^{13}-1)^{13})$, e is chosen as -2. This allows us to perform multiplications of $GF(p)$ elements with positive powers of e, as it heavily takes place in the FFT computation (in lines 7 and 19 of Algorithm 3), with a simple bitwise left rotation, in addition to a simple negation if the power of e is odd.

Algorithm 3: FFT Computation over $GF((2^{13}-1)^{13})$ on MSP430

```
Input: $a(x) = a_0 + a_1x + a_2x^2 \cdots + a_{12}x^{12} \in GF(p^{13})$ where $p = 2^{13}-1$. $R_0, R_1 ... R_6$
       denote used registers. $E_0, E_1 ... E_{12}$ and $O_0, O_1 ... O_{12}$ denote used temporary
       variables.
Output: $(A) = (A_0, A_1, A_2 ... A_{25})$, frequency domain coefficients of $a(x)$ .
1  for $j \leftarrow 0$ to 6 do
2      $R_j \longleftarrow a_{2j}$
3  end
4  $E_0 \longleftarrow R_0 + R_1 + ... + R_6 \bmod p$
5  for $j \leftarrow 1$ to 12 do
6      for $i \leftarrow 1$ to 6 do
7          $R_i \longleftarrow R_i 2^{2i} \bmod p$
8      end
9      $E_j \longleftarrow R_0 + R_1 + ... + R_6 \bmod p$
10 end
11 for $j = 0$ to 5 do
12     $R_j \longleftarrow a_{2j+1}$
13 end
14 $O_0 \longleftarrow R_0 + R_1 + ... + R_5 \bmod p$
15 $A_0 \longleftarrow E_0 + O_0 \bmod p$
16 $A_{13} \longleftarrow E_0 - O_0 \bmod p$
17 for $j = 1$ to 12 do
18     for $i = 0$ to 5 do
19         $R_i \leftarrow R_i 2^{2i+1} \bmod p$
20     end
21     $O_j \leftarrow R_0 + R_1 + ... + R_5 \bmod p$
22     $A_j \leftarrow E_j + (-1)^j O_j \bmod p$
23     $A_{j+13} \longleftarrow E_j - (-1)^j O_j \bmod p$
24 end
25 Return $(A_0, A_1, A_2 ... A_{25})$
```

Note that for $p = 2^{13} - 1$, $e = -2$, $a \in GF\ (p)$ and k a positive integer, the computation $a \times e^k = a \times (-1)^k \times 2^k$ modulo p is equivalent to the simple bitwise left rotation of a by (k mod 13) bits followed by a simple negation if k is odd.

2.1.2.2 Pairwise coefficient multiplication in the frequency domain. The frequency domain multiplication of $GF\ ((2^{13} - 1)^{13})$ elements is carried out through pairwise coefficient multiplications. For (A) and (B), the 26-coefficient frequency domain sequences for $a(x)$ and $b(x)$ in $GF\ ((2^{13} - 1)^{13})$, $r'(x) = a(x)\ b(x) \bmod 2^{13} - 1$ is computed in the frequency domain as

$$R'_j = A_j B_j\ mod(2^{13} - 1)\ , 0 \le j \le 25 \qquad (5)$$

The above 26 coefficient multiplications are the only $GF\ (2^{13} - 1)$ multiplications performed for computing (R') which is dramatically faster than performing 169 coefficient multiplications as needed in schoolbook multiplication. However, (R') needs to be carried back to time domain to complete the $GF\ ((2^{13} - 1)^{13})$ multiplication and find $r(x) = r'(x) \bmod p(x)$. Modular reduction with $p(x)$ becomes very simple when $p(x)$ is selected as $x^{13} - 2$. However, one still needs convert (R') to the time domain polynomial $r'(x)$.

2.1.2.3 Conversion of the product back to the time domain. In order to finalize the finite field multiplication operation in $GF\ ((2^{13} - 1)^{13})$, the 26-element sequence (R') for $r'(x) = a(x)b(x)$ needs to be carried into time domain to realize modular reduction, i.e. $r(x) = r'(x) \bmod x^{13}-2$, efficiently. The conversion can be done using the inverse DFT as follows:

$$r'_i = \frac{1}{26} \sum_{j=0}^{25} R'_j\, e^{-ji} mod\ p\ , 0 \le i \le 25 \qquad (6)$$

Algorithm 4 (Gulen & Baktir, 2020; Gülen & Baktir, 2022) performs the above inverse DFT computation efficiently to convert (R') in the frequency domain to $r\ (x) = a(x).b(x)\ mod\ x^{13} - 2$ in the time domain. The algorithm optimizes equation (6) by utilizing the inverse FFT and by interleaving it with the reduction operation modulo the field generating polynomial $x^{13} - 2$. For the selected finite field $GF((2^{13} - 1)^{13})$, the 26th primitive root of unity e for the inverse FFT computation is chosen as -2. This allows us to perform multiplications of $GF(p)$ elements with negative powers of e, as it heavily takes place in the inverse FFT computation (in lines 7, 17, 27 and 38 of Algorithm 4), with a simple bitwise right rotation, in addition to a simple negation if the power of e is odd.

Algorithm 4: Inverse FFT over $GF((2^{13}-1)^{13})$ on MSP430

Input: $(R') = (R'_0, R'_1, R'_2 ... R'_{25})$, the frequency domain coefficients for $r'(x) = a(x)b(x)$, where $a(x)$ and $b(x)$ are in $GF(p^{13})$ and $p = 2^{13} - 1$. $R_0, R_1 ... R_6$ denote used registers. $E_0, E_1 ... E_{12}$ and $O_0, O_1 ... O_{12}$ denote used temporary variables.

Output: $r(x) = r'(x) \bmod p(x)$ where $p(x) = x^{13} - 2$.

```
1  for $j \leftarrow 0$ to 6 do
2      $R_j \longleftarrow R'_{2j}$
3  end
4  $E_0 \leftarrow R_0 + R_1 + ... + R_6 \bmod p$
5  for $j \leftarrow 1$ to 12 do
6      for $i \leftarrow 1$ to 6 do
7          $R_i \longleftarrow R_i / 2^{2i} \bmod p$
8      end
9      $E_j \leftarrow R_0 + R_1 + ... + R_6 \bmod p$
10 end
11 for $j \leftarrow 7$ to 12 do
12     $R_{j-7} \longleftarrow R'_{2j+1}$
13 end
14 $E_0 \leftarrow E_0 + R_0 + R_1 + ... + R_5 \bmod p$
15 for $j \leftarrow 1$ to 12 do
16     for $i \leftarrow 7$ to 12 do
17         $R_{i-7} \longleftarrow R_{i-7} / 2^{2i} \bmod p$
18     end
19     $E_j \leftarrow E_j + R_0 + R_1 + ... + R_5 \bmod p$
20 end
21 for $j \leftarrow 0$ to 6 do
22     $R_j \longleftarrow R'_{2j+1}$
23 end
24 $O_0 \leftarrow R_0 + R_1 + ... + R_6 \bmod p$
25 for $j \leftarrow 1$ to 12 do
26     for $i \leftarrow 0$ to 6 do
27         $R_i \longleftarrow R_i / 2^{2i+1} \bmod p$
28     end
29     $O_j \leftarrow R_0 + R_1 + ... + R_6 \bmod p$
30 end
31 for $j \leftarrow 7$ to 12 do
32     $R_{j-7} \longleftarrow R'_{2j+1}$
33 end
34 $O_0 \leftarrow O_0 + R_0 + R_1 + ... + R_5 \bmod p$
35 $r_0 \longleftarrow (3E_0 - O_0)7876 \bmod p$
36 for $j \leftarrow 1$ to 12 do
37     for $i \leftarrow 7$ to 12 do
38         $R_{i-7} \longleftarrow R_{i-7} / 2^{2i+1} \bmod p$
39     end
40     $O_j \leftarrow O_j + R_0 + R_1 + ... + R_5 \bmod p$
41     $r_j \longleftarrow (3E_j + O_j(-1)^{j-1})7876 \bmod p$
42 end
43 Return $r_0 + r_1 x + r_2 x^2 + .... r_{12} x^{12}$
```

Note it for $p = 2^{13}-1$, $e = -2$, $a \in GF(p)$ and a positive integer k, the computation $a \times e^{-k} = a \times (-1)^k \times 2^{-k}$ modulo p is equivalent to the simple bitwise right rotation of a by (k mod 13) bits followed by a simple negation if k is odd.

2.1.3 Elliptic curve cryptography using Edwards curve. The main operation in ECC is scalar point multiplication, i.e., computing $s \cdot P$ for an integer s and a point P on the elliptic curve. ECC scalar point multiplication involves performing several ECC point addition and doubling operations. To achieve ECC point multiplication, the binary method (Menezes et al., 1996) can be used, where the bits of the scalar s are scanned one bit at a time starting with the most significant bit, and for each scanned bit, a point doubling operation is performed, in addition to a point addition operation if the scanned bit is 1. However, the binary method is both inefficient and vulnerable against simple power analysis (P. C. Kocher, 1996). As an alternative to the binary method, and in order to help mitigate its drawbacks, the NAF4 and Comb methods can be used for ECC scalar point multiplication of random and fixed points, respectively. NAF4 and Comb require computing a significantly reduced number point additions and doublings compared to the binary method (Koblitz et al., 2004).

Edwards curves, proposed in (Edwards, 2007), are a new form for elliptic curves and defined by the following equation:

$$x^2 + y^2 = c^2(1 + dx^2y^2) \tag{7}$$

The ECC point addition of the two distinct points P_1 and P_2 on an Edwards curve is computed as

$$P_3(x_3, y_3) = P_1(x_1, y_1) + P_2(x_2, y_2)\,, \tag{8}$$
$$where\ \ x_3 = \frac{x_1y_2 + y_1x_2}{c(1 + dx_1x_2y_1y_2)}\ \ and\ y_3 = \frac{y_1y_2 - x_1x_2}{c(1 - dx_1x_2y_1y_2)}$$

The ECC point doubling operation on the point $P_1(x_1, y_1)$ on an Edwards curve is computed as

$$P_2(x_2, y_2) = 2.P_1(x_1, y_1), \tag{9}$$
$$where\ \ x_2 = \frac{2x_1y_1c}{x_1^2 + y_1^2}\ \ and\ y_2 = \frac{(y_1^2 - x_1^2)c}{2c^2 - (x_1^2 + y_1^2)}$$

The above ECC point operations can be achieved in *projective coordinates* (Blake et al., 1999; Enge, 1999; Koblitz et al., 2004) to avoid costly inversions. For $x^2 + y^2 = c^2(1 + dx^2y^2)$, with $c = 1$, d random and $d \cdot c4 \neq 1$, the formulae for ECC point doubling and addition in projective coordinates over prime fields are given in Algorithms 5 and 6, respectively, (Bernstein & Lange, 2007).

Algorithm 5: Elliptic curve point doubling in projective coordinates over prime fields using Edwards curves

```
Input: $P_1(X_1 : Y_1 : Z_1)$
Output: $P_2(X_2 : Y_2 : Z_2) = 2 \cdot P_1$
1  $T_1 \leftarrow X_1, T_2 \leftarrow Y_1, T_3 \leftarrow Z_1$
2  $T_4 \longleftarrow T_1 + T_2$
3  $T_1 \longleftarrow T_1^2$
4  $T_2 \longleftarrow T_2^2$
5  $T_3 \longleftarrow T_3^2$
6  $T_4 \longleftarrow T_4^2$
7  $T_3 \longleftarrow 2 \cdot T_3$
8  $T_5 \longleftarrow T_1 + T_2$
9  $T_2 \longleftarrow T_1 - T_2$
10 $T_4 \longleftarrow T_4 - T_5$
11 $T_3 \longleftarrow T_5 - T_3$
12 $T_1 \longleftarrow T_3 \cdot T_4$
13 $T_3 \longleftarrow T_3 \cdot T_5$
14 $T_2 \longleftarrow T_2 \cdot T_5$
15 $X_2 \leftarrow T_1, Y_2 \leftarrow T_2, Z_2 \leftarrow T_3$
16 Return $(X_2 : Y_2 : Z_2)$
```

Algorithm 6: Elliptic curve point addition in projective coordinates over prime fields using Edwards curves

```
Input: $P_1(X_1 : Y_1 : Z_1)$ and $P_2(X_2 : Y_2 : Z_2)$
Output: $P_3(X_3 : Y_3 : Z_3) = P_1 + P_2$
1  $T_1 \leftarrow X_1$, $T_2 \leftarrow Y_1$, $T_3 \leftarrow Z_1$, $T_4 \leftarrow X_2$,
   $T_5 \leftarrow Y_2$, $T_6 \leftarrow Z_2$
2  $T_3 \longleftarrow T_3 \cdot T_6$
3  $T_7 \longleftarrow T_1 + T_2$
4  $T_8 \longleftarrow T_4 + T_5$
5  $T_1 \longleftarrow T_1 \cdot T_4$
6  $T_2 \longleftarrow T_2 \cdot T_5$
7  $T_7 \longleftarrow T_7 \cdot T_8$
8  $T_7 \longleftarrow T_7 - T_1$
9  $T_7 \longleftarrow T_7 - T_2$
10 $T_7 \longleftarrow T_7 \cdot T_3$
11 $T_8 \longleftarrow T_1 \cdot T_2$
12 $T_8 \longleftarrow d \cdot T_8$
13 $T_2 \longleftarrow T_2 - T_1$
14 $T_2 \longleftarrow T_2 \cdot T_3$
15 $T_3 \longleftarrow (T_3)^2$
16 $T_1 \longleftarrow T_3 - T_8$
17 $T_3 \longleftarrow T_3 + T_8$
18 $T_2 \longleftarrow T_2 \cdot T_3$
19 $T_3 \longleftarrow T_3 \cdot T_1$
20 $T_1 \longleftarrow T_1 \cdot T_7$
21 $X_3 \leftarrow T_1$, $Y_3 \leftarrow T_2$, $Z_3 \leftarrow T_3$
22 Return $(X_3 : Y_3 : Z_3)$
```

2.2 Our ECC Implementations

We implemented ECC and the required arithmetic algorithms on MSP430, low-power 16-bit microcontrollers typically used in WSN applications. As our development environment we used Ingenjörsfirman Anders Rundgren (IAR) Embedded Workbench. We coded our implementation using mostly the assembly language, rather than the C language. Because even when we configured the optimization settings of the C compiler for high speed, the resulting code was still slow. For instance, when we coded in C, one word multiplication took around 130 clock cycles (without hardware multiplier support). Whereas, we realized the same operation with an assembly subroutine, and it executed in only 93 clock cycles. Furthermore, when the C language is used, the C compiler optimizes a word multiplication operation by taking into account the values of the operands, resulting in varying execution times for different operand values. This would make a C code vulnerable against side-channel attacks. By using the assembly language, we were able to write a word multiplication subroutine that is both faster and constant-time. Furthermore, many of the word operations required in

Algorithm 1 can be achieved more efficiently by using the assembly language. For instance, a bitwise rotation operation in *GF* $(2^{13} - 1)$, as required in Algorithm 1, can be implemented faster by using assembly instructions such as the *swap byte* instruction, which are not directly accessible using the C language.

2.2.1 Field arithmetic optimizations. The read and write instructions, for accessing memory, have a significant impact on the efficiency of arithmetic operations. MSP430 has a reduced instruction set computing architecture with only 27 instructions and 7 addressing modes. The addressing mode of an instruction determines its execution time. An instruction takes less clock cycles to execute if the register addressing mode is used; however, there are only 12 general purpose registers available on MSP430. We used these 12 registers for storing our operands as much as possible. By storing frequently used constants in these registers, we were able to eliminate the extra clock cycles.

2.2.1.1 Addition and subtraction. In our ECC implementation *GF* $(2^{13} - 1)$ addition is the most frequently used operation. Note that subtraction in *GF* $(2^{13} - 1)$ is similar to addition, with the exception of an additional *XOR* instruction applied to flip the bits of the subtracted operand. Hence, the cost of subtraction is only 1 clock cycle more than addition. We allocated two registers to store constant values for masking and checking the most significant bit of the operands during these operations. We realized *GF*$(2^{13} - 1)$ addition and subtraction in four and five clock cycles, respectively, as described with Assembly Code 1 and Assembly Code 2 (Gulen & Baktir, 2016).

Assembly Code 1: Addition in $GF(2^{13} - 1)$

```
BIT     R14,R10; (R14 = 0x1000)
RLC     R10
AND     R13,R10; (R13 = 0x1FFF)
```

Assembly Code 2: Subtraction in $GF(2^{13} - 1)$

```
XOR     R13,R14; (R13 = 0x1FFF)
ADD     R15,R14
BIT     R11,R14
ADC     R14
AND     R13,R14
```

2.2.1.1 ***Modular multiplication.*** Efficiency of the modular multiplication over *GF* $(2^{13} - 1)$ directly influences the efficiency of multiplications in *GF* $(2^{13} - 1)^{13}$. In many embedded applications, there are limitations on power consumption and hence, energy efficiency, in addition to timing performance, is an important criterion. For power efficiency, we performed multiplication in *GF* $(2^{13} - 1)$ without using hardware multiplier support. We computed intermediary 26-bit product in *GF* $(2^{13} - 1)$ multiplication in a bit-serial fashion by performing a series of additions, as described with Assembly Code 3, partially. The Assembly Code 3 shows scanning only 2 bits of the operand; however, it is executed repeatedly for all 13 bits, in the original code (Gulen & Baktir, 2016). Here, all the bits of one operand are scanned through, and if the scanned bit is 1, the other operand is added to the partial product, and then, the partial product is shifted to the left. We perform the modular reduction operation after the 26-bit integer product is computed. Our implementation of *GF* $(2^{13} - 1)$ multiplication is a constant time implementation and avoids side-channel attacks by executing additions with 0 to equate clock cycles while scanning bits and branching accordingly. Our *GF* $(2^{13} - 1)$ multiplication code takes 105 clock cycles to execute, where 93 cycles are spent for integer multiplication and 12 cycles for modular reduction.

Assembly Code 3: Multiplication without hardware multiplier support in $GF(2^{13} - 1)$

```
      ...
      RLA      R6
      JC       D1
      RLA      R4
      JMP      C1
D1:   ADD      R7,R4
      RLA      R4
      ADD      0,r4
C1:   RLA      R6
      JC       D2
      RLA      R4
      JMP      C2
D2:   ADD      R7,R4
      RLA      R4
      ...
```

2.2.1.2 Bitwise rotations. Since large number of bitwise rotations are performed over $GF(2^{13} - 1)$ elements in Algorithm 1, we optimized this operation. We frequently made use of the *arithmetic shift* and *shift with carry* instructions which both execute in a single clock cycle. We also used the *set bit*, *test bit*, and swap *byte* instructions to realize rotations by spending the minimal number of clock cycles. We pursued various strategies to reduce the number of required clock cycles for rotations by different numbers of bits. We carried out 1-bit left-rotation by checking the most significant bit of the operand and then shifting it to the left through carry. Here, we performed a mask operation to move the carry bit into the least significant bit position. The 1-bit left-rotation operation, shown in Figure 1 and given with Assembly Code 4 is accomplished in three clock cycles (Gulen & Baktir, 2016). We performed 2,3 and 4-bit left-rotations by repeated 1-bit left rotations. For rotations by more than 4 bits, we utilized the swap byte instruction and exchanged the low and high bytes of the 13-bit operand, as shown in Figure 2. We optimized 5,6,7,8,9, and 10-bit left-rotations through the use of the swap byte instruction and *mask/store* operations. For rotations by different numbers of bits, we reordered and/or modified our code to achieve the best cycle time in each case. Exemplarily, we give Assembly Code 5 for our implementation of 6-bit left-rotation, which takes nine clock cycles. We achieved 1-bit right-rotation by shifting the least significant bit of the operand to the carry flag and then setting the 13th bit of the operand if the carry flag is set. We achieved 12-bit left-rotation through 1-bit right rotation, as implemented with Assembly Code 6. Finally, we achieved 11-bit left-rotation by performing two 12-bit left-rotations.

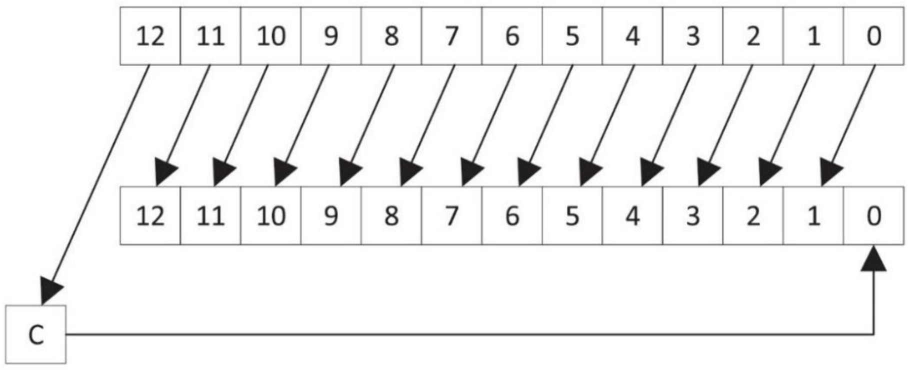


Figure 1. 1-bit left rotation (Gulen & Baktir, 2016).

Assembly Code 4: 1-bit left rotation in $GF(2^{13}-1)$

```
BIT     R14,R10; (R14 = 0x1000)
RLC     R10
AND     R13,R10; (R13 = 0x1FFF)
```

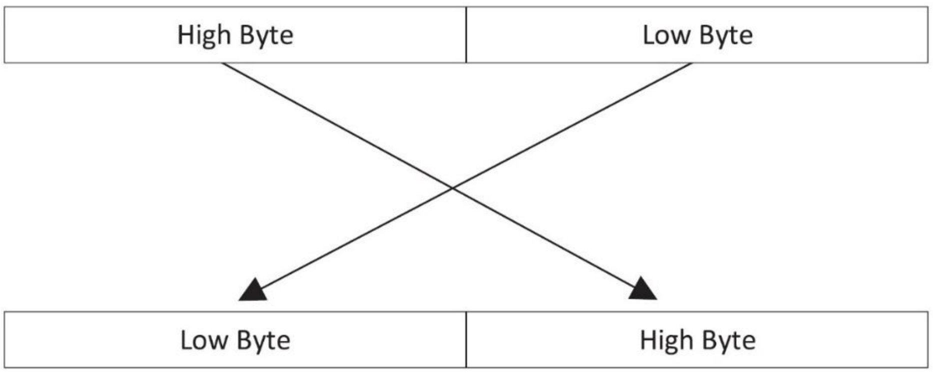


Figure 2. Swap byte instruction (Gulen & Baktir, 2016).

Assembly Code 5: 6-bit left rotation in $GF(2^{13}-1)$

```
MOV     R15,R10; Store operand in R10
AND     R7,R10; (R7 = 0x007F)
SWPB    R10
RRA     R10
RRA     R10
RLA     R15
SWPB    R15
AND     R6,R15; (R6 = 0x003F)
BIS     R10,R15
```

Assembly Code 6: 12-bit left rotation in $GF(2^{13}-1)$

```
        RRA     R10
        JNC     done
        BIS     R14,R10; (R14 = 0x1000)
done:
```

2.2.2 Point arithmetic optimizations.

We improve Algorithms 5 and 6 by taking advantage of FFT based multiplication and squaring operations. Our improved algorithms are given in Algorithms 7 and 8 (Gulen & Baktir, 2020; Gülen & Baktir, 2022).

Algorithm 7 is a reordered and optimized version of Algorithm 5. It takes advantage of FFT based finite field multiplication and squaring computations. In line1 of the algorithm, the FFTs of X_1 and Y_1 are computed, and then added in the frequency domain to find the FFT of $R_1 = X_1 + Y_1$. The computed FFTs of X_1, Y_1 and R_1 are stored. The stored frequency domain representations of X_1, Y_1 and R_1 are used in lines $2-4$ (marked bold) for the three finite field squarings. Please note that for these three finite field squarings, a total number of only two forward FFT computations are performed, i.e., $FFT(X_1)$ and $FFT(Y_1)$ in line 1, instead of three as required in Algorithm 2. Furthermore, in line 10, the computed FFT of Z_1 is stored and reused in line 11 (marked bold). Similarly, in line 11, the computed FFT of R_2 is stored and reused in line 12 (marked bold). Please note that each time the stored result of an FFT computation is reused, a forward FFT computation is saved in Algorithm 2.

Algorithm 7: Elliptic curve point doubling in projective coordinates over prime fields using Edwards curves and FFT based multiplication/squaring

```
Input: $P = (X_1 : Y_1 : Z_1)$, $R_1$ and $R_2$ are temporary registers.
Output: $2P = (X_2 : Y_2 : Z_2)$
1  $R_1 \longleftarrow FFT(X_1) + FFT(Y_1)$   $//FFTs\ stored$
2  $R_1 \longleftarrow \mathbf{R}_1{}^2$
3  $X_1 \longleftarrow \mathbf{X}_1{}^2$
4  $Y_1 \longleftarrow \mathbf{Y}_1{}^2$
5  $Z_1 \longleftarrow 2Z_1{}^2$
6  $R_2 \longleftarrow X_1 + Y_1$
7  $Y_1 \longleftarrow X_1 - Y_1$
8  $R_1 \longleftarrow R_1 - R_2$
9  $Z_1 \longleftarrow R_2 - Z_1$
10 $X_2 \longleftarrow Z_1 \cdot R_1$   $//FFT\ of\ Z_1\ stored$
11 $Z_2 \longleftarrow \mathbf{Z}_1 \cdot R_2$   $//FFY\ of\ R_2\ stored$
12 $Y_2 \longleftarrow Y_1 \cdot \mathbf{R}_2$
13 Return $(X_2 : Y_2 : Z_2)$
```

Algorithm 8: Elliptic curve point addition in projective coordinates over prime fields using Edwards curves and FFT based multiplication/squaring

```
Input: $P = (X_1 : Y_1 : Z_1)$ , $Q = (X_2 : Y_2 : Z_2)$ , $R_1$ and $R_2$ are temporary registers.
Output: $P + Q = (X_3 : Y_3 : Z_3)$
1  $Z_1 \longleftarrow Z_1 \cdot Z_2$
2  $R_1 \longleftarrow FFT(X_1) + FFT(Y_1)$   $//FFTs\ stored$
3  $R_2 \longleftarrow FFT(X_2) + FFT(Y_2)$   $//FFTs\ stored$
4  $R_1 \longleftarrow \mathbf{R}_1 \cdot \mathbf{R}_2$
5  $X_1 \longleftarrow \mathbf{X}_1 \cdot \mathbf{X}_2$
6  $Y_1 \longleftarrow \mathbf{Y}_1 \cdot \mathbf{Y}_2$
7  $R_1 \longleftarrow R_1 - X_1$
8  $R_1 \longleftarrow R_1 - Y_1$
9  $R_1 \longleftarrow R_1 \cdot Z_1$   $//FFT\ of\ Z_1\ stored$
10 $R_2 \longleftarrow d \cdot X_1 \cdot Y_1$   $//FFTs\ of\ X_1\ and\ Y_1\ stored$
11 $Y_1 \longleftarrow \mathbf{Y}_1 - \mathbf{X}_1$   $//FFT\ of\ Y_1\ stored$
12 $Y_1 \longleftarrow \mathbf{Y}_1 \cdot \mathbf{Z}_1$
13 $Z_1 \longleftarrow \mathbf{Z}_1{}^2$
14 $X_1 \longleftarrow Z_1 - R_2$
15 $Z_1 \longleftarrow Z_1 + R_2$
16 $Y_3 \longleftarrow Y_1 \cdot Z_1$   $//FFT\ of\ Z_1\ stored$
17 $Z_3 \longleftarrow \mathbf{Z}_1 \cdot X_1$   $//FFT\ of\ X_1\ stored$
18 $X_3 \longleftarrow \mathbf{X}_1 \cdot R_1$
19 Return $(X_3 : Y_3 : Z_3)$
```

Algorithm 8 is a reordered and optimized version of Algorithm 6. It takes advantage of FFT based finite field multiplication and squaring computations. In lines $2 - 3$ of the algorithm, the FFTs of X_1, X_2, Y_1 and Y_2 are computed and stored. Only two addition operations are performed in the frequency domain on the stored FFTs to readily obtain the FFTs of $R_1 = X_1 + Y_1$ and $R_2 = X_2 + Y_2$. The FFTs of R_1 and R_2 are also stored. The stored FFTs of R_1, R_2, X_1, X_2, Y_1 and Y_2 are readily used in lines $4-6$ (denoted with bold color) for the three finite field multiplication computations. Thus, for three finite field multiplications, a total number of only four forward FFT computations are performed, instead of six as required in Algorithm 2. Furthermore, in lines $11-13$ of the algorithm, the stored FFTs of Y_1, X_1 and Z_1 are reused (marked bold). Similarly, in line 16, the FFT of Z_1 is computed and stored. The stored FFT of Z_1 is reused in line 17 (marked bold). Likewise, in line 17, the FFT of X_1 is computed and stored, and reused in line 18 (marked bold).

2.3 Performance Evaluation and Comparison

We give in Table 1 the timings for our FFT based multiplication and squaring operations in $GF((2^{13}-1)^{13})$ as well as the timings in (Gulen & Baktir, 2016) for DFT Montgomery multiplication. While DFT based Montgomery multiplication in $GF((2^{13}-1)^{13})$ takes 1.18 ms, our FFT based multiplication implementation takes 1.3 ms which is 10.2% slower. However, our FFT based squaring implementation takes only 1.06 ms while the squaring operation using DFT based Montgomery multiplication takes the same time as multiplication, i.e. 1.18 ms. Hence, FFT based squaring is shown to be 11.3% faster than squaring using DFT Montgomery multiplication (GULEN & BAKTIR, 2022).

We implement ECC scalar point multiplication, the main operation for encryption/decryption in ECC, using the NAF4 method for multiplication of random points and the Comb method for multiplication of fixed points. We achieve ECC scalar point multiplication in 1.74 s for random points and 0.88 s for fixed points. In (Gulen & Baktir, 2016), the same operation, in the same setting but by using DFT Montgomery multiplication, was achieved in 1.97 s and 0.98 s for random and fixed points. As given with Table 1, this work achieves 11% and 10% better timings for random and fixed point multiplication operations, respectively.

A summary of our timing results for ECC point multiplication and the underlying elliptic curve point operations and arithmetic operations, as well as the timing results of (Gulen & Baktir, 2016), are given in Table 1. Our ECC implementation utilizes FFT based multiplication (Algorithms 2, 3 and 4) for squaring and multiplication, whereas (Gulen & Baktir, 2016) uses DFT based Montgomery multiplication (Algorithm 1).

Table 1.

Timings for ECC with DFT Based Montgomery Multiplication vs. FFT Based Multiplication.

Operations	Timings
ECC with DFT Montgomery Multiplication (Gulen & Baktir, 2016)	
DFT based Montgomery multiplication	1.18 ms
Edwards Curve Point Doubling	8.52 ms
Edwards Curve Point Addition	14.49 ms
Edwards Curve Random Point Multiplication with NAF4	1.97 s
Edwards Curve Fixed Point Multiplication with Comb4	0.98 s
ECC with FFT Based Multiplication/Squaring (GULEN & BAKTIR, 2022)	
FFT Based Multiplication/Squaring (Algorithm 2)	1.3 ms / 1.06 ms
Improved Edwards Curve Point Doubling (Algorithm 7)	7.55 ms
Improved Edwards Curve Point Addition (Algorithm 8)	12.95 ms
Edwards Curve Random Point Multiplication with NAF4	1.74 s
Edwards Curve Fixed Point Multiplication with Comb4	0.88 s

Note that while DFT based Montgomery multiplication is faster than FFT based multiplication, FFT based squaring is even faster. In our ECC random point multiplication implementation over $GF\ ((2^{13} - 1)^{13})$ with NAF4 , significantly more point doublings than point additions are performed. On average around 37 point additions and 170 point doublings are performed (Koblitz et al., 2004). Hence, the performance of ECC point doubling is the determining factor in the performance of ECC random point multiplication. In our ECC point doubling implementation (Algorithm 7), four squarings and three multiplications in $GF\ ((2^{13} - 1)^{13})$ are performed. Since squaring with FFT based multiplication (Algorithms 2, 3 and 4) is faster than squaring with DFT based Montgomery multiplication (Algorithm 1), ECC random point multiplication is faster when FFT based multiplication is used.

Similarly, in our ECC fixed point multiplication implementation over $GF((2^{13}-1)^{13})$, around 42 point doublings and 39 point additions are performed (Koblitz et al., 2004). Hence, the performance of ECC point doubling is the determining factor in the performance of ECC random point multiplication. Since more squarings than multiplications are performed in ECC point doubling (Algorithm 5) and squaring with FFT based multiplication (Algorithms 2, 3 and 4) is faster than squaring with DFT based Montgomery multiplication (Algorithm 1), our ECC fixed point multiplication implementation with FFT based multiplication is faster than the previous work which uses DFT Montgomery multiplication.

2.3.1 Power consumption comparison of our works. We investigate the energy efficiency of our implementation and compare it with the previous implementations. In order to obtain energy measurements, we run our codes on the experimenter board MSP-EXP430FG4618 which has an MSP430FG4618 microcontroller onboard (Texas Instruments, 2018). Since our ECC implementation is for the basic 1-series MSP430F1611, we can run our implementation on the experimenter board without changing our code. Using the flash emulation tool (MSP-FET) (Texas Instruments, 2020)and the Power Log feature of the IAR Embedded Workbench development tool, we are able to obtain energy measurements (IAR Systems, 2020).

The average power consumption for our ECC random point multiplication implementation in this work is 17.3 mW without using the hardware multiplier. The work in (Gulen & Baktir, 2020) uses the hardware multiplier and achieves the same operation with a power consumption figure of 17.66 mW. The average power consumption for our ECC fixed point multiplication implementation in this work is 17.5 mW without using the hardware multiplier. The work in (Gulen & Baktir, 2020) uses the hardware multiplier and achieves the same operation with a power consumption figure of 17.88 mW. For ECC random and fixed point multiplication operations, we achieve around 2% improved power efficiency. The previous work in (Gulen & Baktir, 2020), which utilizes the hardware multiplier unit of the MSP430 microcontroller, has faster timings on the experimental board, i.e. 1.35 s and 0.64 s for random and fixed point multiplication, respectively. Since these execution times are less than the execution times in the proposed implementation, the total energy consumptions are also lower. The total energy consumptions for the implementations in (Gulen & Baktir, 2020) are 23.84 mWs and 11.44 mWs for random and fixed point multiplication, respectively. While these total energy consumption figures are better than those of the proposed implementation, the average power consumption figures for the proposed implementation are better. Note that we run our ECC implementations on the MSP430FG4618 microcontroller without using its hardware multiplier and obtain our energy/power measurements on it. While this helps us gain power/energy efficiency in terms of dynamic power usage, the microcontroller still uses static power due to its onboard hardware multiplier. We could only use this microcontroller for power/energy measurements because it is the microcontroller contained in the experimental board with the FET emulator which we use to obtain timing and power/energy measurements. We believe that better energy/power efficiency could be achieved on another version of MSP430, such as MSP430F2274 or MSP430G2955, which does not have a hardware multiplier. We would like to note that low-cost, low-power MPS430 microcontroller versions do not have a hardware multiplier unit. For instance, from the low-power 1-series MSP430 versions, only the microcontrollers with the device names MSP430x14x and MSP430x16x have a hardware multiplier unit. Whereas, other low-power 1-series MSP430 versions, such as MSP430F1122, MSP430F1232, MSP430F135 and MSP430F155, do not have an onboard hardware multiplier (Texas Instruments, 2004). Among other series of MSP430 microcontrollers, there are also models without a hardware multiplier unit. One such example is MSP430F2274 which is equipped in the Texas Instrument ez430-RF2500 wireless module that is designed to be deployed in wireless sensor network applications (Texas Instruments, 2015). Unlike the ECC implementation in (Gulen & Baktir, 2020), our ECC implementation, which does not require a hardware multiplier, has the additional advantage of being able to run efficiently also on these extremely constrained microcontrollers without a hardware multiplier. The main motivation for using a processor without a hardware multiplier would be to increase the battery lifetime or for applications where sensor nodes harvest their own energy and need to operate under extremely low power constraints.

We also compare our work against the previous work in (Gulen & Baktir, 2016) in terms of power efficiency. Note that both works implement ECC without using the hardware multiplier. This work uses the FFT (Algorithm 2), whereas the work in (Gulen & Baktir, 2016) uses DFT Montgomery multiplication (Algorithm 1) for finite field multiplication. Our work achieves ECC scalar point multiplication with a power consumption figure of 17.3 mW for random points and 17.5 mW for fixed points. In (Gulen & Baktir, 2016), the same operation, in the same setting but by using DFT Montgomery multiplication, was achieved with a power consumption figure of 18.2 mW for both random and fixed points. Hence, this work achieves 5% and 4% better power efficiency for random and fixed point multiplication, respectively. Furthermore, our work achieves ECC scalar point multiplication with an energy consumption of 29.81 mWs for random points and 15.27 mWs for fixed points. In (Gulen & Baktir, 2016), the same operation, in the same setting but by using DFT Montgomery multiplication, was achieved with the energy consumption of 34.97 mWs and 17.36 mWs for random and fixed points. Hence, this work achieves 15% and 12% better energy efficiency for random and fixed point multiplication, respectively. A summary of all the energy/power measurements for our implementations of ECC point multiplication and the underlying elliptic curve point operations and arithmetic operations, as well as the energy/power measurements for the ECC implementation in (Gulen & Baktir, 2016), are given in Table 2.

Table 2.

Power Measurements for ECC with DFT Based Montgomery Multiplication vs. FFT Based Multiplication.

Operations	Total Energy	Average Power
ECC with DFT Montgomery Multiplication (Gulen & Baktir, 2016)		
DFT based Montgomery multiplication	0.021 mWs	18.2 mW
Edwards Curve Point Doubling	0.15 mWs	18.2 mW
Edwards Curve Point Addition	0.25 mWs	18.2 mW
Edwards Curve Random Point Multiplication with NAF4	34.97 mWs	18.2 mW
Edwards Curve Fixed Point Multiplication with Comb4	17.36 mWs	18.2 mW
ECC with FFT Based Multiplication/Squaring (GULEN & BAKTIR, 2022)		
FFT Based Multiplication/Squaring (Algorithm 2)	0.022/0.018 mWs	17.1/17.2 mW
Improved Edwards Curve Point Doubling (Algorithm 7)	0.13 mWs	17.2 mW
Improved Edwards Curve Point Addition (Algorithm 8)	0.22 mWs	17.2 mW
Edwards Curve Random Point Multiplication with NAF4	29.81 mWs	17.3 mW
Edwards Curve Fixed Point Multiplication with Comb4	15.27 mWs	17.5 mW

2.3.2 Performance comparison with others' works. In Table 3, we present our timings for ECC random point multiplication on the MSP430F1611 as well as the timings of the related work in the literature on the same microcontroller. Liu et al.'s work, which uses a 159-bit Montgomery curve, presents the fastest timing for random point multiplication on the MSP430 microcontroller (Z. Liu et al., 2010). They use the Montgomery ladder method and achieve random point multiplication in 3,460,000 clock cycles which is equivalent to 0.48 s at 8 MHz clock frequency. Gouvêa et al.'s work, which uses the 160-bit curve *secp160r1* that has a slightly smaller elliptic curve group order than ours, achieves ECC random point multiplication in 0.58 s (Gouvêa et al., 2012). Our previous ECC implementation over *GF* $((2^{13}-1)^{13})$ on the MSP430F149, a similar microcontroller to the MSP430F1611, achieves random point multiplication in 1.55 s (Gülen & Baktir, 2014). Please note that our ECC random point multiplication implementation in this work, which exploits the NTT-based finite field multiplication/squaring and the FFT, is more than 18% faster than our previous implementation on the same elliptic curve. Wang et al.'s implementation of elliptic curve random point multiplication over a 160-bit elliptic curve has a timing value of 3.51 s which is significantly worse than our timing result (H. Wang et al., 2006). In a later work, the same authors improve their timing to 1.60 s; however, their new implementation is still 22% slower than our work (H. Wang & Li, 2006).

Table 3.

Timings for ECC Random Point Multiplication.

Microcontroller	Field	Method	Hardware Multiplier	Timing
(Liu, et al., 2016) MSP430F1611 @8 MHz	F_{P159}	Montgomery ladder	YES	0.48 s
(Gouvea, et al., 2012) MSP430F1611 @8 MHz	F_{P160}	NAF4	YES	0.58 s
(Gulen & Baktir, 2020) MSP430F1611 @8 MHz	$F_{(2^{13}-1)^{13}}$	NAF4	YES	1.31 s
(Gulen & Baktir, 2014) MSP430F149 @8 MHz	$F_{(2^{13}-1)^{13}}$	NAF4	YES	1.55 s
(Wang & Li) MSP430F1611 @8 MHz	F_{P160}	-	YES	1.60 s
(GULEN & BAKTIR, 2022) MSP430F1611 @8 MHz	$F_{(2^{13}-1)^{13}}$	NAF4	NO	1.74 s
(Gulen & Baktir, 2016) MSP430F149 @8 MHz	$F_{(2^{13}-1)^{13}}$	NAF4	NO	1.97 s
(Wang, et al., 2006) MSP430F1611 @8 MHz	F_{P160}	-	YES	3.51 s

Please note that the timing figure for our ECC implementation is for a 169-bit elliptic curve with a higher security level, whereas the others' works use the smaller ordered 159-bit and 160-bit elliptic curves.

In Table 4, we present our timings for ECC fixed point multiplication on the MSP430F1611 as well as the timings of the related work in the literature on the same microcontroller. Liu et al.'s work, which uses a 159-bit twisted Edwards curve, presents the fastest timing for fixed point multiplication on the MSP430 microcontroller (Z. Liu et al., 2010).

Table 4.
Timings for ECC Fixed Point Multiplication.

Microcontroller	**Field**	**Method**	**Hardware Multiplier**	**Timing**
(Liu, et al., 2016) MSP430F1611 @ 8 MHz	F_{P159}	Comb	YES	0.24 s
(Gouvea, et al., 2012) MSP430F1611 @ 8 MHz	F_{P160}	NAF4	YES	0.52 s
(Gulen & Baktir, 2020) MSP430F1611 @ 8 MHz	$F_{(2^{13}-1)^{13}}$	Comb4	YES	0.65 s
(Szczechowiak et al., 2008) MSP430F1611 @ 8 MHz	F_{P160}	Comb	YES	0.72 s
(Gulen & Baktir, 2022) MSP430F1611 @ 8 MHz	$F_{(2^{13}-1)^{13}}$	Comb4	NO	0.88 s
(Gulen & Baktir, 2016) MSP430F1611 @ 8 MHz	$F_{(2^{13}-1)^{13}}$	Comb4	NO	0.98 s
(Szczechowiak et al., 2008) MSP430F1611 @ 8 MHz	$F_{2^{163}}$	Comb	YES	1.04 s
(Wenger & Werner, 2011) MSP430F1611 @ 8 MHz	F_{P160}	-	YES	1.09 s
(Wang et al., 2006) MSP430F1611 @ 8 MHz	F_{P160}	Sliding Window	YES	1.44 s
(Liu & Ning) MSP430F1611 @ 8 MHz	F_{P160}	Sliding window	YES	1.58 s

They use the Comb method and twisted Edwards curves to achieve fixed point multiplication in 1,920,000 clock cycles which is equivalent to 0.24 s at 8 MHz clock frequency. Gouvêa et al.'s work, which uses the 160-bit elliptic curve *secp160r1* and the 4NAF method, achieves ECC fixed point multiplication in 0.52 s (Gouvêa et al., 2012). Liu et al.'s timing for 160-bit ECDSA signature generation (considered to have around the same timing value as elliptic curve fixed point multiplication) is 1.58 s, which is twice slower than our implementation that uses a larger 169-bit elliptic curve. Wang et al.'s work on the same microcontroller achieves elliptic curve fixed point multiplication in 1.44 s over a 160-bit elliptic curve. Wenger's implementation of elliptic curve fixed point multiplication on a 160-bit elliptic curve takes 8,779,931 clock cycles which is equivalent to 1.09 s at 8 MHz clock frequency (Wenger & Werner, 2011). Szczechowiak et al.'s work achieves elliptic curve fixed point multiplication in 0.72 s using a 160-bit elliptic curve over a prime field (Szczechowiak et al., 2008) and in 1.04 s using a 163-bit elliptic curve over a binary field (Szczechowiak et al., 2008). Our timing for elliptic curve fixed point multiplication over a larger ordered 169-bit elliptic curve is slightly better than their results. Please note that the timing figure for our ECC implementation is for a 169-bit elliptic curve with a higher security level, whereas the others' works use the smaller ordered 159-bit, 160-bit and 163-bit elliptic curves.

2.4 Chapter Conclusion

We implemented ECC on the MSP430 microcontrollers with and without using hardware multiplier support and using number theoretic transform based finite field arithmetic. We showed that ECC can be run efficiently on extremely constrained devices when FFT based squaring and multiplication operations are used. Since FFT based squaring and multiplication require dramatically fewer word multiplications, we discarded the hardware multiplier supported by MSP430 microcontrollers. Instead of utilizing the hardware multiplier, we realized a fixed-time word multiplication subroutine with addition and shift operations. Thus, our ECC implementations are also suitable for power-critical applications. We realized ECC point multiplication in 0.89 s and 1.74 s for fixed and random points, which are 10% and 13% faster, respectively, in comparison with the previous work in (Gulen & Baktir, 2016). Moreover, the total energy consumption of our ECC implementation for fixed and random point multiplication is 12% and 15% less than the previous implementation. In our proof-of-concept implementation, we realized ECC without using the hardware multiplier on the MSP430FG4618 microcontroller and obtained energy/power measurements on it. We achieved power/energy savings by not using the available onboard hardware multiplier and thus by eliminating dynamic power consumption due to the use of the hardware multiplier. We used this microcontroller because it is the microcontroller contained in the FET emulator which we use to obtain our timing and energy/power measurements. By using the MSP430 versions MSP430F2274, MSP430G2955, MSP430F1122, MSP430F1232, MSP430F135 or MSP430F155, which do not contain an onboard hardware multiplier, static power/energy consumption due to the hardware multiplier can also be eliminated, and thus better power/energy efficiency could be achieved by using our proposed algorithms and implementations.

Chapter 3
RSA Cryptography for Wireless Sensor Nodes

RSA cryptosystem, which is the first general-purpose PKC algorithm, is by far also the most widely deployed one (Dimitrov et al., 2022; Fotohi et al., 2020; Fu et al., 2021; Ganbaatar et al., 2021; Jiao et al., 2020; Karim et al., 2021; Lin et al., 2018; Medha Nag et al., 2020; Ochoa-Jimenez et al., 2020; Pavani & Sriramya, 2021; Vollala et al., 2017; Wahab et al., 2021; H. Yu & Kim, 2020). 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 (Z. Liu et al., 2010). 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 (Barker, 2020). While the same security level is reached with ECC utilizing a shorter key and hence smaller computational load (Gulen & Baktir, 2020; Wenger & Werner, 2011), 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 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 (Koblitz & Menezes, 2016). 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 (Barker, 2020). 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 (Karray et al., 2018). Similarly, the sensor nodes WSN1120L, WSN1120CL, WSN1101ANL and WSN1101ACL of WiSense use the MSP430G2955 microcontroller (WiSense Technologies, 2019).

3.1 Background

The RSA public key cryptosystem (Rivest et al., 1978), introduced by Rivest, Shamir and Adleman in 1978, is the first ever and still most widely used general purpose public key cryptographic algorithm. In RSA, encryption and decryption are described by the below equations where x is the plaintext, y is the ciphertext, e is the public encryption key, N is part of the public key and d is the private (decryption) key.

$$RSA\ Encryption : y = x^e \ mod\ N \tag{10}$$

$$RSA\ Decryption : x = y^d \ mod\ N \tag{11}$$

The modulus N, which is used for performing modular arithmetic, is the product of two large primes, denoted with p and q (Rivest et al., 1978). Furthermore, there is the following relationship between e and d:

$$e = d^{-1}\ mod\ \varphi(N) \tag{12}$$

where $\varphi(N)$, the Euler's Phi function, has the following value:

$$\varphi(N) = (p-1) \times (q-1) \tag{13}$$

If the public key N can be factorized into p and q, it would be possible to compute $\varphi(N) = (p-1) \times (n-1)$ and obtain the secret decryption key d by computing $e^{-1}\ mod\ \varphi(N)$. Hence, the security of RSA relies on the difficulty of factorizing the modulus *N*. If *N* is large enough, it cannot be factorized to obtain d and thus RSA cannot be broken.

3.1.1 Sliding Window Method.

The exponentiation operation for RSA decryption can be achieved by using the basic binary scan method (Menezes et al., 1996). This method scans the bits of the exponent one bit at a time starting with the most significant non-zero bit. For the most significant non-zero bit, the intermediary result is initially set to the value of the base. Then, for each new bit scanned, the intermediary result is updated with its square. If the newly scanned bit is 1, then the intermediary result is further updated by multiplying it with the base value. All arithmetic operations, i.e., multiply and square, are performed in modulo the RSA modulus *N*.

Algorithm 9: Modular Exponentiation with the 4-bit Sliding Window Method

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

In this method, the number of square operations conducted is equal to one less than the number of bits in the exponent. The number of multiply operations is equal to one less than the Hamming weight of the exponent, which, on average, is half the bit-length of the exponent minus 1. Hence, for a *t*-bit RSA decryption operation, basic binary scan method performs $t-1$ modular squarings and on average $(t-1)/2$ modular multiplications.

The sliding window method for exponentiation improves upon the binary scan method by reducing the number of required modular multiplications. In Algorithm 9 (Gulen et al., 2019), the sliding window method is given for a 4-bit window. The base and the exponent are represented with b and e, respectively.

3.1.2 Chinese Remainder Theorem. Applying the CRT is a common method used to speed up RSA decryption (Menezes et al., 1996). Using the CRT, for any integer value y, y mod N can be uniquely represented as (y_p, y_q) where y_p and y_q are the residues of y modulo the relatively prime numbers p and q, respectively. Note that the CRT representation can be used in RSA decryption due to the fact that the RSA modulus N is the product of two prime numbers. The normal integer representation of $y \bmod N$ can be recovered from its CRT representation using the formula

$$y = (q \times c_p) \times y_p + (p \times c_q) \times y_q mod\ N\ , \tag{14}$$

where

$$c_p = \ q^{-1}\ mod\ p\ , c_q = \ p^{-1}\ mod\ q\ . \tag{15}$$

Modular multiplication, as it takes place in RSA decryption, can be done in the CRT representation using the moduli p and q, instead of the usual RSA modulus N. Furthermore, repeated multiplications modulo N, as it takes place in RSA decryption, can be performed in the CRT representation and the final result can be converted back to the normal integer representation. Hence, the RSA decryption operation, i.e., the computation of $x = y^d \, mod \, N$ can be achieved using the CRT as

$$x_p = y_p^{d_p} mod \, p \, , \tag{16}$$

$$x_q = y_q^{d_q} mod \, q \, , \tag{17}$$

$$x = (q \times c_p) \times x_p + (p \times c_q) \times x_q \, mod \, N \tag{18}$$

where $d_p = d \, mod \, (p-1)$ and $d_q = d \, mod \, (q-1)$. Note that the computations of $x_p = y_p^d$ and $x_q = y_q^d$ are simplified as $y_p^{d_p}$ and $y_q^{d_q}$, respectively, by taking advantage of the Euler's theorem which states that $y_p^d = y_p^{d \, mod \, \varphi(p)} \, (mod \, p)$ and $y_q^d = y_q^{d \, mod \, \varphi(p)} \, (mod \, q)$.

In the above RSA decryption operation using the CRT, the values d_p, d_q, $q \times c_p$, $p \times c_q$, c_p and c_q can be precomputed. Furthermore, in (16) and (17), note that the bit-lengths of the operands that are exponentiated, as well as the exponents, are half their sizes in the normal RSA decryption without the CRT. Therefore, using the CRT reduces the overall timing of RSA decryption dramatically by a factor of up to four.

3.1.3 Montgomery Multiplication. Since arithmetic is performed in modulo *N* for RSA encryption and decryption, the result of every multiplication operation needs to be reduced modulo *N*. The Montgomery multiplication algorithm (Montgomery, 1985), given in Algorithm 10 (Gulen et al., 2019), can be used to achieve multiplication modulo *N* efficiently. For multiplying the two integers *A* and *B* with the algorithm, they should be converted into their Montgomery forms as $\bar{A}$ and $\bar{B}$, where $\bar{A} = A \times r \, mod \, N$ and $\bar{B} = B \times r \, mod \, N$. Here the constant *r* can be picked to be any integer that is greater than and relatively prime with *N*. For purposes of efficiency, *r* can be chosen to be the smallest power of 2 that is greater than *N*. Montgomery multiplication of $\bar{A}$ and $\bar{B}$ produces $\bar{A} \times \bar{B} \times r^{-1} \, mod \, N$ which is the

Montgomery form $\bar{Z}$ of the product $Z = A \times B \ mod \ N$. Thus, Montgomery multiplication preserves the Montgomery form and repeated multiplications, such as the multiplications that make up an exponentiation, can be performed using Montgomery multiplication.

The arithmetic operations performed for achieving Montgomery multiplication are integer multiplication, addition and subtraction. Subtraction is computed only when the intermediary result T_1 is larger than or equal to *N*. The cost of the division operation by 2^n, on line 3 of Algorithm 10, can be ignored since the first *n* bits of T_1 are zeroes and the higher bits readily give the result.

Algorithm 10: Montgomery Modular Multiplication

```
Input: $\bar{A} = A \cdot r \bmod N$ and $\bar{B} = B \cdot r \bmod N$
    where $r = 2^n$ and $n = \lceil \log_2 N \rceil$
Output: $\bar{C} = A \cdot B \cdot r \bmod N$
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: $(\bar{Z} \leftarrow T_1)$
```

3.1.4 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, 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 (Karatsuba, 1963). An illustration of Karatsuba-Ofman for multiplying the *n*-bit integers *A* and *B* is given in Figure 3. As given with Figure 3, 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 $O(m^2)$ to $O(m^{\log_2 3})$ for the multiplication of two m word integers.

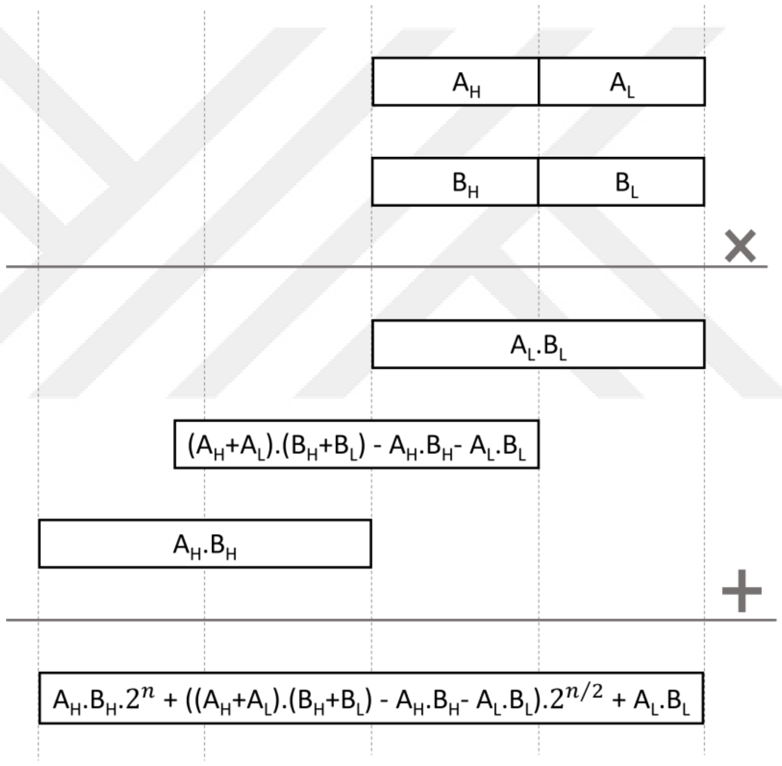
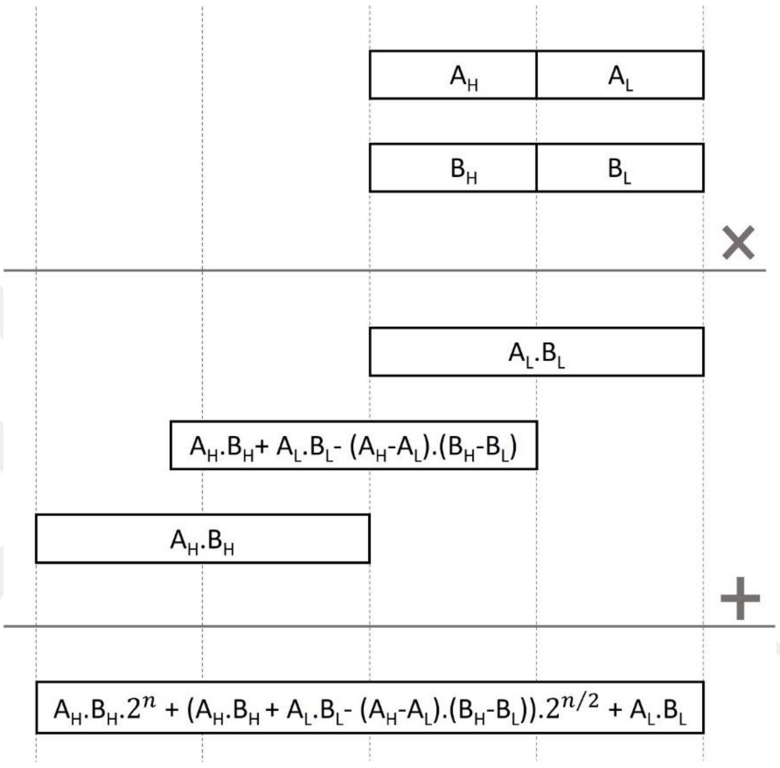

A_H
A_L
B_H
B_L
×
$A_L.B_L$
$(A_H+A_L).(B_H+B_L) - A_H.B_H - A_L.B_L$
$A_H.B_H$
+
$A_H.B_H.2^n + ((A_H+A_L).(B_H+B_L) - A_H.B_H - A_L.B_L).2^{n/2} + A_L.B_L$

Figure 3. Karatsuba-Ofman multiplication.

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 (Hinterwälder et al., 2015; Hutter & Schwabe, 2015). With this approach, the two halves of the integers to be multiplied are subtracted from each other, instead of being added. Figure 4 shows the operations that take place in subtractive Karatsuba-Ofman.



A_H
A_L
B_H
B_L
×
$A_L.B_L$
$A_H.B_H + A_L.B_L - (A_H-A_L).(B_H-B_L)$
$A_H.B_H$
+
$A_H.B_H.2^n + (A_H.B_H + A_L.B_L - (A_H-A_L).(B_H-B_L)).2^{n/2} + A_L.B_L$

Figure 4. Subtractive Karatsuba-Ofman multiplication.

3.2 Our RSA Implementation on MSP430

We implement 2048-bit RSA on three target MSP430 microcontrollers, namely MSP430F1611, MS430F2618 and MSP430F5529 (Texas Instruments, 2004). Since these three generations of MSP430 support the same instruction set, we are able to use the same code for the microcontrollers with little modifications. 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.

All the acceleration techniques described in Chapter 3.1 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 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 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 Chapter 3.2.1 and 3.2.2.

The techniques which we use in our RSA decryption/encryption implementation are summarized in Figure 5. 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 Chapter 3.2.4, are used to mitigate side-channel attacks and protect the private decryption key. Finally, the sliding window method is used for RSA decryption.

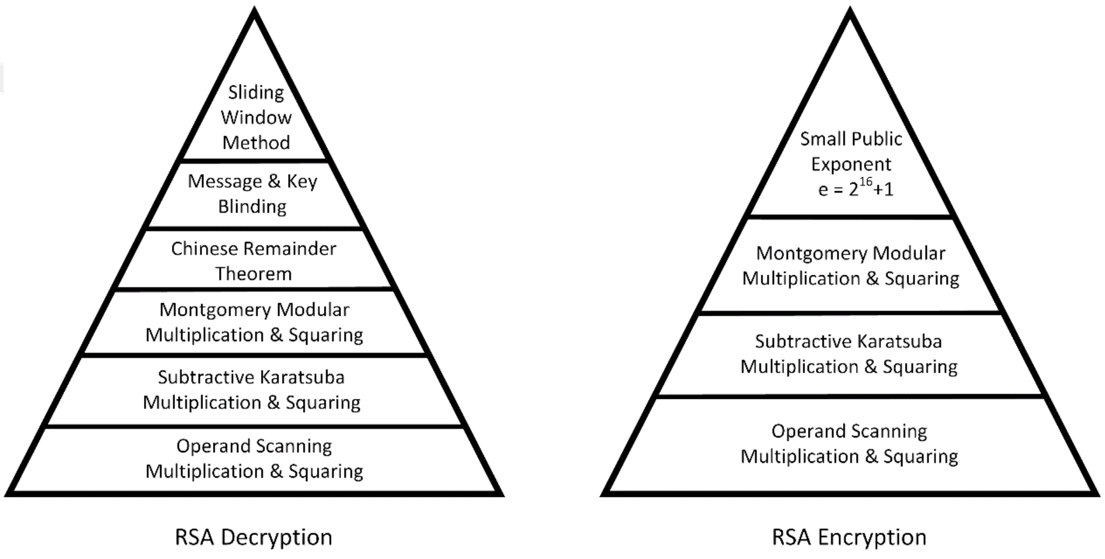

Sliding Window Method
Message & Key Blinding
Chinese Remainder Theorem
Montgomery Modular Multiplication & Squaring
Subtractive Karatsuba Multiplication & Squaring
Operand Scanning Multiplication & Squaring
RSA Decryption
Small Public Exponent e = 2^{16}+1
Montgomery Modular Multiplication & Squaring
Subtractive Karatsuba Multiplication & Squaring
Operand Scanning Multiplication & Squaring
RSA Encryption

Figure 5. Techniques used in the proposed RSA implementation according to their performance impact.

3.2.1 Fixed time subtractive Karatsuba-Ofman Multiplication. 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 4, 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×FFFF or 0×0000 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 *XOR*ing 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×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, *XOR*ing $(A_H - A_L)$ 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×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.

Subroutine 1: Assembly subroutine for A$_H$-A$_L$ computation in 128-bit subtractive Karatsuba multiplication.

```
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 2: Assembly subroutine for B$_H$-B$_L$ 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]
```

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 $T_2[0:7]$, $T_1[0:7]$ and $T_0[0:7]$, 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 $T_1[0:7]$ and the carry-out bit is stored in SWA.

As seen in Figure 4, 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 $T_2[0:7]$ and $T_0[0:7]$, 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.

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

```
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 4: Subroutine for adding t_2+t_0-t_1 to finalize 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]
```

3.2.2 Fixed time subtractive Karatsuba-Ofman Squaring. 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, 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 10. In Montgomery multiplication, three integer multiplications are performed, as shown in lines 1, 2 and 3 of Algorithm 10. 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 10 is a squaring. We speed up this integer squaring by optimizing our subtractive Karatsuba-Ofman multiplication implementation for the squaring computation.

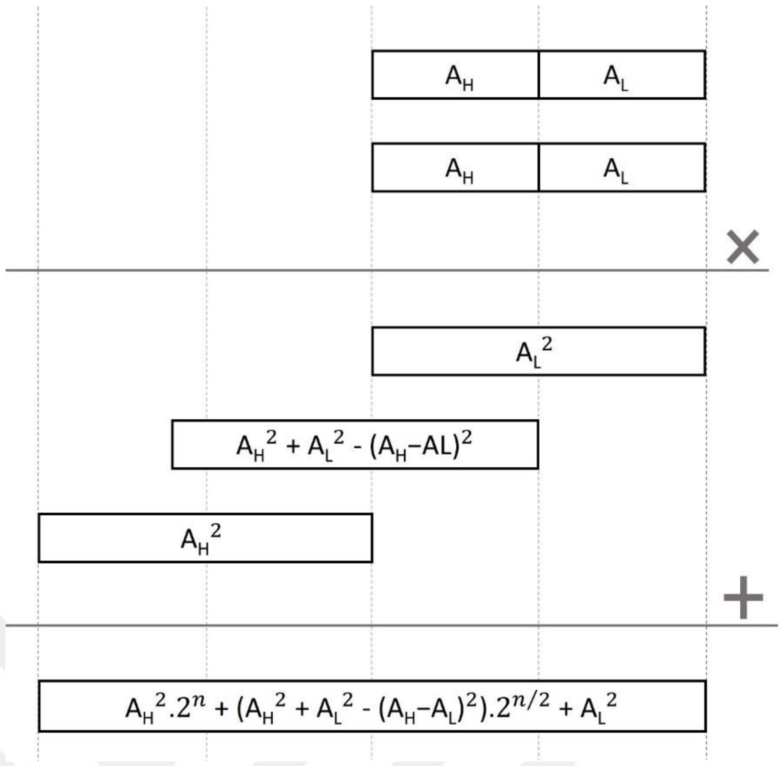

A_H
A_L
A_H
A_L
×
A_L^2
$A_H^2 + A_L^2 - (A_H–AL)^2$
A_H^2
+
$A_H^2.2^n + (A_H^2 + A_L^2 - (A_H–A_L)^2).2^{n/2} + A_L^2$

Figure 6. Subtractive Karatsuba-Ofman squaring.

The operations that take place in our subtractive Karatsuba-Ofman squaring implementation are depicted in Figure 6. 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 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 T_0[0 : 7], T_1[0 : 7] and T_2[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 T_1[0 : 7] and the sign word CW. We represent this result as T_1[0 : 7] || CW, the concatenation of T_1[0 : 7] and CW. We then subtract $T_1 - T_0 - T_2$, stored in T_1[0 : 7] || CW, from T_0[4 : 7] || T_2[0 : 4] in order to compute the middle 128-bits of the result. We add the generated carry word to T_2[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.

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

```
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 6: Subroutine for subtracting t_1-t_0-t_2 to finalize 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]
```

3.2.3 Fixed time operand scanning methods. 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.

For our implementation on MSP430F5529, which has a 32×32-bit onboard hardware multiplier, we implement 2048-bit 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, i.e., $T_2 = A_H \cdot$

B_H, $T_1 = (A_H - A_L) \cdot (B_H - B_L)$, and $T_0 = A_L \cdot B_L$, and the method 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, to compute ($A_1 \cdot A_2 + A_2 \cdot A_1$), 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 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 7. 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.

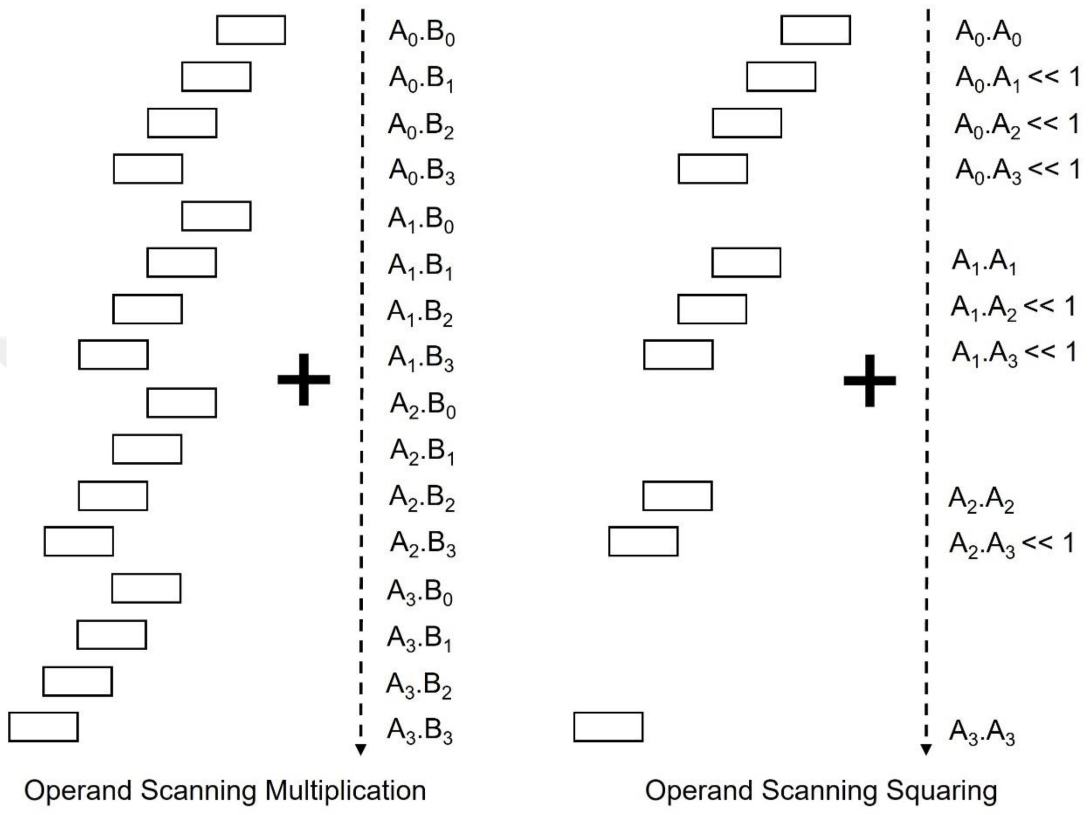
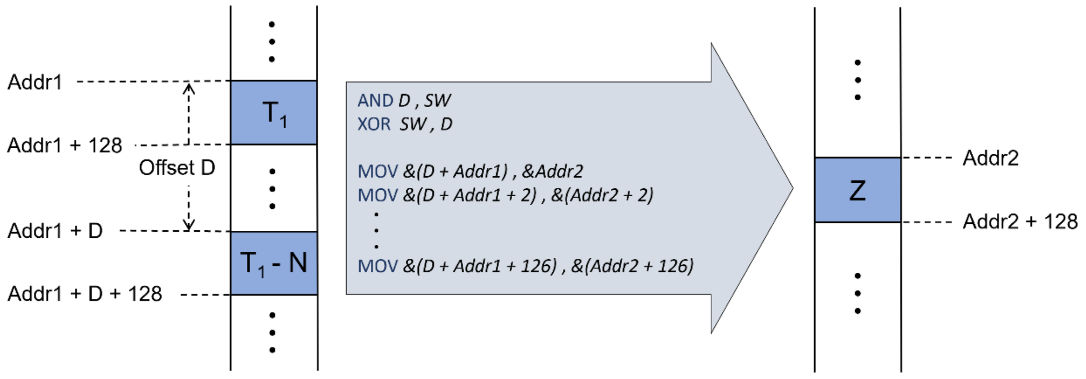


Figure 7. *64-bit operand scanning multiplication and squaring method using 16x16 hardware multiplier.*

3.2.4 Side-Channel Countermeasures. Side-channel attacks on cryptographic implementations vary in a wide range and several low-cost countermeasures have been proposed to mitigate them (Duan et al., 2016; Gao et al., 2019; Jurecek et al., 2019; Kannwischer et al., 2018; Petrvalsky et al., 2016; Xu et al., 2018). The first attack that comes to mind is the SPA attack (P. C. Kocher, 1996). In RSA decryption, exponentiation using the binary 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 with the naked eye by looking at the power consumption trace of RSA decryption under an oscilloscope, the secret key would be revealed easily. An effective solution we use in our implementation is the sliding window method which ensures similar power profiles for different secret exponent values.

Another vulnerability against SPA 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 10. 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×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×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 8.

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 (P. Kocher et al., 2011; P. C. Kocher, 1996). For affordable DPA attack mitigation, we use the blinding method (P. C. Kocher, 1996; Z. Liu et al., 2010). We blind both the ciphertext and the secret key.



Addr1
Addr1 + 128
Offset D
Addr1 + D
Addr1 + D + 128
T1
T1 - N
AND D , SW
XOR SW , D
MOV &(D + Addr1) , &Addr2
MOV &(D + Addr1 + 2) , &(Addr2 + 2)
MOV &(D + Addr1 + 126) , &(Addr2 + 126)
Z
Addr2
Addr2 + 128

Figure 8. Fixed time branch implementation for 1024-bit Montgomery multiplication.

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

$$V_i = (V_f^{-1})^d mod\ N \tag{19}$$

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 (P. C. Kocher, 1996). 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 works (P. C. Kocher, 1996; Z. Liu et al., 2010). Exponent blinding is basically the randomization of the private decryption key *d* described as

$$d' = d + r \cdot \varphi(N) \tag{20}$$

where r is a random integer and $\varphi(N) = (p-1)\cdot(q-1)$. We apply exponent blinding to our RSA decryption implementation and thus our CRT exponents become $d'_p = d_p + r \cdot \varphi(p)$ and $d'_q = d_q + r \cdot \varphi(q)$ where $\varphi(p) = (p-1)$, $\varphi(q) = (q-1)$ and r is a random 32-bit integer as suggested in (Z. Liu et al., 2010). Exponent blinding increases the length of the exponent by 32 bits which results in a timing overhead of around 3% in decryption.

3.3 Performance Evaluation and Comparison. We implement 1024-bit and 2048-bit RSA on three target MSP430 microcontrollers, namely MSP430F5529, MSP430F2618, and MSP430F1611. We develop our RSA implementations in C language. In addition, we write assembly subroutines for implementing core arithmetic operations. Our timings for 1024-bit RSA encryption and decryption are given in Table 5 (Gulen et al., 2019). We compare our timings with the existing 1024-bit RSA implementations in the literature on the same or similar microcontrollers. Our implementation for 1024-bit RSA encryption has the best timings among all, as seen in Table 6.

Table 5.

Our Timings for 1024-bit RSA Operations on the MSP430 Microcontroller.

RSA 1024-bit Operation	Microcontroller	# Clock Cycles	Time (sec)
Encryption	MSP430F5529 @25 MHz	1,189,089	0.047
Encryption	MSP430F2618 @16 MHz	1,611,482	0.10
Encryption	MSP430F1611 @8 MHz	1,743,445	0.21
Decryption	MSP430F5529 @25 MHz	28,608,119	1.14
Decryption	MSP430F2618 @16 MHz	40,007,873	2.50
Decryption	MSP430F1611 @8 MHz	43,368,720	5.42

Table 6.

Timing Comparisons for RSA Implementation with a 1024-Bit Key on Constrained microcontrollers

RSA 1024-bit Operation	**Microcontroller**	**Clock Cycles**	**Time (s)**
Encryption (Gulen et al., 2019)	MSP430F1611 @8 MHz	1,743,445	0.21
Encryption (Qiu et al., 2017)	MSP430F1611 @8 MHz	3,665,144	0.45
Encryption (Gura et al., 2004)	ATmega128 @8 MHz	-	0.43
Encryption (H. Wang & Li, 2006)	ATmega128 @8 MHz	-	0.79
Encryption (Gura et al., 2004)	CC1010 @14 MHz	-	4.48
Decryption (Gulen et al., 2019)	MSP430F1611 @8 MHz	43,368,720	5.42
Decryption (Qiu et al., 2017)	MSP430F1611 @8 MHz	44,639,340	5.58
Decryption (Z. Liu et al., 2010)	ATmega128 @8 MHz	75,680,000	9.46
Decryption (Gura et al., 2004)	ATmega128 @8 MHz	-	10.99
Decryption (H. Wang & Li, 2006)	ATmega128 @8 MHz	-	21.5
Decryption (Gura et al., 2004)	CC1010 @14 MHz	-	106.66

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 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 7 and present the timings of our 2048-RSA implementation on the three target MSP430 microcontrollers.

Table 7.

Our Timings for 2048-bit RSA Operations 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

There are several existing implementations of RSA with a 1024-bit key on MSP430 or similar constrained microcontrollers (Gulen et al., 2019; Gura et al., 2004; Z. Liu et al., 2010; Qiu et al., 2017; H. Wang & Li, 2006). 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 (Gura et al., 2004). To make a fair evaluation, we compare with the work in (Gura et al., 2004) 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 in (Gura et al., 2004) as shown in Table 8.

Table 8.

Timing Comparisons for RSA Implementation with a 2048-Bit Key on Constrained Microcontrollers.

RSA 2048-bit Operation	Microcontroller	Memory Usage (Data & Code)	Time (s)
Encryption [Ours]	MSP430F1611 @8 MHz	1.5 kB & 8.2 kB	0.67
Encryption (Gura et al., 2004)	ATmega128 @8 MHz	1.3 kB & 2.8 kB	1.94
Decryption [Ours]	MSP430F1611 @8 MHz	3.8 kB & 13 kB	34.90
Decryption (Gura et al., 2004)	ATmega128 @8 MHz	1.8 kB & 7.7 kB	83.36

Our 2048-bit RSA implementation utilizes the same techniques as those used in (Gura et al., 2004), 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. Moreover, we unroll our recursive implementation to avoid loop overheads. 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.

3.4 Chapter Conclusion

RSA is the oldest and the most adopted public-key cryptographic algorithm that is utilized by the existing Internet infrastructure and related applications. We presented a practical, side-channel resistant implementation of 1024-bit and 2048-bit RSA on the constrained microcontrollers that are widely used in WSN nodes and IoT devices. Our fastest RSA implementation achieved 2048-bit encryption and decryption in 0.14 s and 7.56 s. 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 sliding window, ciphertext blinding and secret key blinding methods.

Chapter 4
Conclusions

In this thesis, we present and use novel techniques for implementing public-key cryptography for embedded constrained devices. The constrained devices, i.e. microcontrollers, are used in WSN and IoT applications widely and they have limited resources i.e. CPU speed and memory capacity. Additionally, such microcontrollers commonly powered by small batteries with low capacity, or they harvest the energy from the ambient sources in the environment e.g., solar, thermal, vibration. Considering the limitations of such devices, applying computational expensive and complex PKC operations is challenging. In our studies we applied various accelerations methods while implementing the two most popular PKC, namely ECC and RSA, for MSP430 microcontrollers. With our combined optimization methods, we achieved better timing results on MSP430 microcontrollers, compared to existing other works in the literature which use same or similar microcontrollers.

We realized our ECC implementations using NTT and we utilized an OEF $GF(p^m)$, using Edwards curve on projective coordinates. NTT benefits from convolution theorem and reduces the number of the required base field $GF(p)$ multiplications which is the core optimization method for our ECC implementations. In our studies, we use DFT based Montgomery multiplication and FFT based multiplication for faster multiplications to accelerate ECC scalar point multiplications. Our studies presented in the thesis are the first software implementations using NTT for ECC, in the literature.

Compared to ECC, RSA requires a larger key size, at least 1024-bit. It is usually considered as a drawback of the RSA to implement on the constrained microcontrollers. However, RSA can have other advantages, e.g., allowing fast digital signature verification. The common acceleration methods such as Chinese remainder theorem, sliding window method and small public exponent improve RSA performance significantly. Moreover, we utilized subtractive Karatsuba-Ofman multiplication technique with optimized operand scanning method to achieve even faster RSA encryption/decryption timings. Thus, our RSA timings for the MSP430 microcontrollers are the fastest in the literature compared to similar microcontrollers, for 1024-bit key size and 2048-bit key size. Additionally, we applied existing side-channel attack countermeasures in our RSA implementations to mitigate SPA and DPA.

One of the possible future research topics, after the thesis is, implementing ECC for larger key size on the constrained microcontrollers. We believe, use of optimized NTT based computations can be more impactful, with the larger ECC key size, in terms of performance, especially for the constrained devices. Power efficiency is another important criterion for the constrained embedded devices besides the timing performance. Thus, comparative investigations of the power/energy efficiencies of different arithmetic algorithms and cryptographic implementations on the constrained embedded devices are promising research directions.

REFERENCES

Adu-Manu, K. S., Adam, N., Tapparello, C., Ayatollahi, H., & Heinzelman, W. (2018). Energy-harvesting wireless sensor networks (EH-WSNs): A review. In *ACM Transactions on Sensor Networks* (Vol. 14, Issue 2). https://doi.org/10.1145/3183338

Akyildiz, I. F., Melodia, T., & Chowdhury, K. R. (2007). A survey on wireless multimedia sensor networks. *Computer Networks*, *51*(4). https://doi.org/10.1016/j.comnet.2006.10.002

Bailey, D. v., & Paar, C. (2001). Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, *14*(3). https://doi.org/10.1007/s001450010012

Baktir, S., & Sunar, B. (2006). Finite field polynomial multiplication in the frequency domain with application to elliptic curve cryptography. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *4263 LNCS*. https://doi.org/10.1007/11902140_103

Baktır, S., Kumar, S., Paar, C., & Sunar, B. (2007). A state-of-the-art elliptic curve cryptographic processor operating in the frequency domain. *Mobile Networks and Applications*, *12*(4). https://doi.org/10.1007/s11036-007-0022-4

Baktır, S., & Sunar, B. (2008a). *Frequency domain finite field arithmetic for elliptic curve cryptography* [PhD]. Worcester Polytechnic Institute.

Baktır, S., & Sunar, B. (2008b). Optimal extension field inversion in the frequency domain. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *5130 LNCS*. https://doi.org/10.1007/978-3-540-69499-1_5

Barker, E. (2020). Recommendation for key management. *NIST Special Publication 800-57*.

Baronti, P., Pillai, P., Chook, V. W. C., Chessa, S., Gotta, A., & Hu, Y. F. (2007). Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. In *Computer Communications* (Vol. 30, Issue 7). https://doi.org/10.1016/j.comcom.2006.12.020

Baylis, J. (1988). Introduction to finite fields and their applications, by R. Lidl and H.

Niederreiter. Pp 407. £19·50. 1986. ISBN 0-521-30706-6 (Cambridge University Press). *The Mathematical Gazette*, *72*(462). https://doi.org/10.2307/3619969

Bernstein, D. J., & Lange, T. (2007). Faster addition and doubling on elliptic curves. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *4833 LNCS*. https://doi.org/10.1007/978-3-540-76900-2_3

Blake, I., Seroussi, G., & Smart, N. (1999). Elliptic Curves in Cryptography. In *Elliptic Curves in Cryptography*. https://doi.org/10.1017/cbo9781107360211

Buchanan, W. J., Li, S., & Asif, R. (2017). Lightweight cryptography methods. *Journal of Cyber Security Technology*, *1*(3–4). https://doi.org/10.1080/23742917.2017.1384917

Chandrakasan, A., Amirtharajah, R., Cho, S. H., Goodman, J., Konduri, G., Kulik, J., Rabiner, W., & Wang, A. (1999). Design considerations for distributed microsensor systems. *Proceedings of the Custom Integrated Circuits Conference*. https://doi.org/10.1109/cicc.1999.777291

Chen, X., Makki, K., Yen, K., & Pissinou, N. (2009). Sensor network security: A survey. *IEEE Communications Surveys and Tutorials*, *11*(2). https://doi.org/10.1109/SURV.2009.090205

Chong, C. Y., & Kumar, S. P. (2003). Sensor networks: Evolution, opportunities, and challenges. *Proceedings of the IEEE*, *91*(8). https://doi.org/10.1109/JPROC.2003.814918

Cooley, J. W., & Tukey, J. W. (1965). An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, *19*(90). https://doi.org/10.2307/2003354

Diffie, W., Diffie, W., & Hellman, M. E. (1976). New Directions in Cryptography. *IEEE Transactions on Information Theory*, *22*(6). https://doi.org/10.1109/TIT.1976.1055638

Dimitrov, V., Vigneri, L., & Attias, V. (2022). Fast Generation of RSA Keys Using Smooth Integers. *IEEE Transactions on Computers*, *71*(7). https://doi.org/10.1109/TC.2021.3095669

Duan, X., Cui, Q., Wang, S., Fang, H., & She, G. (2016). Differential power analysis attack and efficient countermeasures on PRESENT. *Proceedings of 2016 8th IEEE International Conference on Communication Software and Networks,* *ICCSN 2016*. https://doi.org/10.1109/ICCSN.2016.7586627

Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A. H., & Schwabe, P. (2015). High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes, and Cryptography*, *77*(2–3). https://doi.org/10.1007/s10623-015-0087-1

Edwards, H. M. (2007). A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, *44*(3). https://doi.org/10.1090/S0273-0979-07-01153-6

Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., & Uhsadel, L. (2007). A survey of lightweight-cryptography implementations. In *IEEE Design and Test of Computers* (Vol. 24, Issue 6). https://doi.org/10.1109/MDT.2007.178

Elgamal, T. (1985). A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, *31*(4). https://doi.org/10.1109/TIT.1985.1057074

Enge, A. (1999). Elliptic Curves and Their Applications to Cryptography. In *Elliptic Curves and Their Applications to Cryptography*. https://doi.org/10.1007/978-1-4615-5207-9

Fotohi, R., Firoozi Bari, S., & Yusefi, M. (2020). Securing Wireless Sensor Networks Against Denial-of-Sleep Attacks Using RSA Cryptography Algorithm and Interlock Protocol. *International Journal of Communication Systems*, *33*(4). https://doi.org/10.1002/dac.4234

Fu, Y., Wang, W., Meng, L., Wang, Q., Zhao, Y., & Lin, J. (2021). VIRSA: Vectorized In-Register RSA Computation with Memory Disclosure Resistance. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 12918 LNCS*. https://doi.org/10.1007/978-3-030-86890-1_17

Ganbaatar, G., Nyamdorj, D., Cichon, G., & Ishdorj, T. O. (2021). Implementation of RSA cryptographic algorithm using SN P systems based on HP/LP neurons. In *Journal of Membrane Computing* (Vol. 3, Issue 1). https://doi.org/10.1007/s41965-021-00073-3

Gao, S. P., Guo, Y., Aung, Z. T., & Guo, Y. X. (2019). Analysis of Information Leakage from MCU using Neural Network. *EMC COMPO 2019 - 2019 12th International Workshop on the Electromagnetic Compatibility of Integrated Circuits*. https://doi.org/10.1109/EMCCompo.2019.8919889

Gouvêa, C. P. L., & López, J. (2009). Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 5922 LNCS*. https://doi.org/10.1007/978-3-642-10628-6_17

Gouvêa, C. P. L., Oliveira, L. B., & López, J. (2012). Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller. *Journal of Cryptographic Engineering*, *2*(1). https://doi.org/10.1007/s13389-012-0029-z

Gulen, U., Alkhodary, A., & Baktir, S. (2019). Implementing rsa for wireless sensor nodes. *Sensors (Switzerland)*, *19*(13). https://doi.org/10.3390/s19132864

Gülen, U., & Baktir, S. (2014). Elliptic curve cryptography on constrained microcontrollers using frequency domain arithmetic. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *8584 LNCS*(PART 6). https://doi.org/10.1007/978-3-319-09153-2_37

Gulen, U., & Baktir, S. (2016). Elliptic-curve cryptography for wireless sensor network nodes without hardware multiplier support. *Security and Communication Networks*, *9*(18), 4992–5002. https://doi.org/10.1002/sec.1670

Gulen, U., & Baktir, S. (2020). Elliptic curve cryptography for wireless sensor networks using the number theoretic transform. *Sensors (Switzerland)*, *20*(5). https://doi.org/10.3390/s20051507

Gülen, U., & Baktir, S. (2022). FFT enabled ECC for WSN nodes without hardware multiplier support. *Turkish Journal of Electrical Engineering and Computer Sciences*, *30*(1), 94–108. https://doi.org/10.3906/elk-2009-95

Gungor, V. C., Lu, B., & Hancke, G. P. (2010). Opportunities and challenges of wireless sensor networks in smart grid. *IEEE Transactions on Industrial Electronics*, *57*(10). https://doi.org/10.1109/TIE.2009.2039455

Gura, N., Patel, A., Wander, A., Eberle, H., & Shantz, S. C. (2004). Comparing elliptic curve cryptography and RSA on 8-Bit CPUs. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 3156*. https://doi.org/10.1007/978-3-540-28632-5_9

He, D., Kumar, N., & Chilamkurti, N. (2015). A secure temporal-credential-based mutual authentication and key agreement scheme with pseudo identity for wireless sensor networks. *Information Sciences*, *321*. https://doi.org/10.1016/j.ins.2015.02.010

Hinterwälder, G., Moradi, A., Hutter, M., Schwabe, P., & Paar, C. (2015). Full-size high-security ECC implementation on MSP430 microcontrollers. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *8895*. https://doi.org/10.1007/978-3-319-16295-9_2

Hutter, M., & Schwabe, P. (2015). Multiprecision multiplication on AVR revisited. *Journal of Cryptographic Engineering*, *5*(3). https://doi.org/10.1007/s13389-015-0093-2

IAR Systems. (2020). *IDE Project Management and Building Guide*.

Ingemarsson, I., Tang, D. T., & Wong, C. K. (1982). A Conference Key Distribution System. *IEEE Transactions on Information Theory*, *28*(5). https://doi.org/10.1109/TIT.1982.1056542

Jiao, K., Ye, G., Dong, Y., Huang, X., & He, J. (2020). Image Encryption Scheme Based on a Generalized Arnold Map and RSA Algorithm. *Security and Communication Networks*, *2020*. https://doi.org/10.1155/2020/9721675

Jurecek, M., Bucek, J., & Lorencz, R. (2019). Side-Channel Attack on the A5/1 Stream Cipher. *Proceedings - Euromicro Conference on Digital System Design, DSD 2019*. https://doi.org/10.1109/DSD.2019.00099

Kannwischer, M. J., Genêt, A., Butin, D., Krämer, J., & Buchmann, J. (2018). Differential power analysis of XMSS and SPHINCS. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *10815 LNCS*. https://doi.org/10.1007/978-3-319-89641-0_10

Karatsuba, A. (1963). Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 595–596.

Karim, R., Rumi, L. S., Ashiqul Islam, M., Kobita, A. A., Tabassum, T., & Sagar Hossen, M. (2021). Digital signature authentication for a bank using asymmetric key cryptography algorithm and token based encryption. In *Lecture Notes on Data Engineering and Communications Technologies* (Vol. 53). https://doi.org/10.1007/978-981-15-5258-8_79

Karray, F., Jmal, M. W., Garcia-Ortiz, A., Abid, M., & Obeid, A. M. (2018). A comprehensive survey on wireless sensor node hardware platforms. In *Computer Networks* (Vol. 144). https://doi.org/10.1016/j.comnet.2018.05.010

Koblitz, N. (1987). Eliptic curve cryptosystems. *Mathematics of Computation, 48 (177).*

Koblitz, N., & Menezes, A. (2016). A riddle wrapped in an Enigma. *IEEE Security and Privacy*, *14*(6). https://doi.org/10.1109/MSP.2016.120

Koblitz, N., Menezes, A., & Vanstone, S. (2004). Guide to Elliptic Curve Cryptography. In *Guide to Elliptic Curve Cryptography*. https://doi.org/10.1007/b97644

Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 1109*. https://doi.org/10.1007/3-540-68697-5_9

Kocher, P., Jaffe, J., Jun, B., & Rohatgi, P. (2011). Introduction to differential power analysis. *Journal of Cryptographic Engineering*, *1*(1). https://doi.org/10.1007/s13389-011-0006-y

Li, M., Lou, W., & Ren, K. (2010). Data security and privacy in wireless body area networks. *IEEE Wireless Communications*, *17*(1). https://doi.org/10.1109/MWC.2010.5416350

Lin, X. J., Sun, L., & Qu, H. (2018). An efficient RSA-based certificateless public key encryption scheme. *Discrete Applied Mathematics*, *241*. https://doi.org/10.1016/j.dam.2017.02.019

Liu, A., & Ning, P. (2008). TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. *Proceedings - 2008 International Conference on Information Processing in Sensor Networks, IPSN 2008*. https://doi.org/10.1109/IPSN.2008.47

Liu, Z., Großschädl, J., & Kizhvatov, I. (2010). Efficient and Side-Channel Resistant RSA Implementation for 8-bit AVR Microcontrollers. *Workshop on the Security of the Internet of Things - (SecIoT'10).*

MacKenzie, P., Patel, S., & Swaminathan, R. (2010). Password-authenticated key exchange based on RSA. *International Journal of Information Security*, *9*(6). https://doi.org/10.1007/s10207-010-0120-3

McEliece, R. J. (1993). Finite Fields for Computer Scientists and Engineers. In *IEEE Transactions on Information Theory* (Vol. 39, Issue 1). https://doi.org/10.1109/TIT.1993.1603956

Medha Nag, K. G., Sharvari, Vaishnavi, D. v., Rajashree, S., & Honnavalli, P. B. (2020). RSA Implementation on Sensor Data in Cold Storage Warehouse. *2nd IEEE Eurasia Conference on IOT, Communication and Engineering 2020, ECICE 2020*. https://doi.org/10.1109/ECICE50847.2020.9301979

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of applied cryptography. In *Handbook of Applied Cryptography*. https://doi.org/10.2307/2589608

Mentens, N., Batina, L., & Baktır, S. (2015). An elliptic curve cryptographic processor using edwards curves and the number theoretic transform. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *9024*. https://doi.org/10.1007/978-3-319-21356-9_7

Miller, V. (1986). Advances in Cryptology — CRYPTO '85 Proceedings. In *Advances in Cryptology — CRYPTO '85 Proceedings* (Vol. 218).

Montgomery, P. L. (1985). Modular multiplication without trial division. *Mathematics of Computation*, *44*(170). https://doi.org/10.1090/s0025-5718-1985-0777282-x

Mouha, N. (2015). The Design Space of Lightweight Cryptography. *NIST Lightweight Cryptography Workshop 2015*.

Ochoa-Jimenez, E., Rivera-Zamarripa, L., Cruz-Cortes, N., & Rodriguez-Henriquez, F. (2020). Implementation of RSA Signatures on GPU and CPU Architectures. *IEEE Access*, *8*. https://doi.org/10.1109/ACCESS.2019.2963826

Ozdemir, S., & Xiao, Y. (2009). Secure data aggregation in wireless sensor networks: A comprehensive overview. *Computer Networks*, *53*(12). https://doi.org/10.1016/j.comnet.2009.02.023

Pavani, K., & Sriramya, P. (2021). Enhancing public key cryptography using RSA, RSA-CRT and N-Prime RSA with multiple keys. *Proceedings of the 3rd International Conference on Intelligent Communication Technologies and Virtual Mobile Networks, ICICV 2021*. https://doi.org/10.1109/ICICV50876.2021.9388621

Perrig, A., Szewczyk, R., Tygar, J. D., Wen, V., & Culler, D. E. (2002). SPINS:

Security protocols for sensor networks. *Wireless Networks*, *8*(5). https://doi.org/10.1023/A:1016598314198

Petrvalsky, M., Richmond, T., Drutarovsky, M., Cayrel, P. L., & Fischer, V. (2016). Differential power analysis attack on the secure bit permutation in the McEliece cryptosystem. *2016 26th International Conference Radioelektronika, RADIOELEKTRONIKA 2016*. https://doi.org/10.1109/RADIOELEK.2016.7477382

Pollard, J. M. (1971). The fast Fourier transform in a finite field. *Mathematics of Computation*, *25*(114). https://doi.org/10.1090/s0025-5718-1971-0301966-0

Pottie, G. J., & Kaiser, W. J. (2000). Wireless integrated network sensors. *Communications of the ACM*, *43*(5). https://doi.org/10.1145/332833.332838

Qiu, L., Liu, Z., Geovandro, G. C., & Seo, H. (2017). Implementing RSA for sensor nodes in smart cities. *Personal and Ubiquitous Computing*, *21*(5). https://doi.org/10.1007/s00779-017-1044-y

Rader, C. M. (1972). Discrete Convolutions via Mersenne Transforms. *IEEE Transactions on Computers*, *C–21*(12). https://doi.org/10.1109/T-C.1972.223497

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, *21*(2). https://doi.org/10.1145/359340.359342

Roman, R., Alcaraz, C., & Lopez, J. (2007). A survey of cryptographic primitives and implementations for hardware-constrained sensor network nodes. *Mobile Networks and Applications*, *12*(4). https://doi.org/10.1007/s11036-007-0024-2

Szczechowiak, P., Oliveira, L. B., Scott, M., Collier, M., & Dahab, R. (2008). NanoECC: Testing the limits of elliptic curve cryptography in sensor networks. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *4913 LNCS*. https://doi.org/10.1007/978-3-540-77690-1_19

Texas Instruments. (2004). *MSP430x11x2, MSP430F22x2, MSP430G2x55 Mixed Signal Microcontroller Datasheet*.

Texas Instruments. (2015). *eZ430-RF2500 Development Tool User's Guide*.

Texas Instruments. (2018). *MSP430FG4618/F2013 Experimenter Board User's Guide*.

Texas Instruments. (2020). *MSP Debuggers User's Guide*.

Vollala, S., Varadhan, V. v., Geetha, K., & Ramasubramanian, N. (2017). Design of RSA processor for concurrent cryptographic transformations. *Microelectronics Journal*, *63*. https://doi.org/10.1016/j.mejo.2017.03.009

Wahab, O. F. A., Khalaf, A. A. M., Hussein, A. I., & Hamed, H. F. A. (2021). Hiding data using efficient combination of RSA cryptography, and compression steganography techniques. *IEEE Access*, *9*. https://doi.org/10.1109/ACCESS.2021.3060317

Wang, H., & Li, Q. (2006). Efficient implementation of public key cryptosystems on mote sensors. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *4307 LNCS*. https://doi.org/10.1007/11935308_37

Wang, H., Sheng, B., & Li, Q. (2006). Elliptic curve cryptography-based access control in sensor networks. *International Journal of Security and Networks*, *1*(3–4). https://doi.org/10.1504/ijsn.2006.011772

Wang, Y., Attebury, G., & Ramamurthy, B. (2006). A survey of security issues in wireless sensor networks. In *IEEE Communications Surveys and Tutorials* (Vol. 8, Issue 2). https://doi.org/10.1109/COMST.2006.315852

Wenger, E., & Werner, M. (2011). Evaluating 16-bit processors for elliptic curve cryptography. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7079 LNCS*. https://doi.org/10.1007/978-3-642-27257-8_11

WiSense Technologies. (2019). *WSN1120L, WSN1120CL, WSN1101ANL, WSN1101ACL Datasheet*.

Xu, J., Fan, A., Lu, M., & Shan, W. (2018). Differential Power Analysis of 8-Bit Datapath AES for IoT Applications. *Proceedings - 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications and 12th IEEE International Conference on Big Data Science and Engineering, Trustcom/BigDataSE 2018*. https://doi.org/10.1109/TrustCom/BigDataSE.2018.00205

Ye, W., Heidemann, J., & Estrin, D. (2002). An energy-efficient MAC protocol for wireless sensor networks. *Proceedings - IEEE INFOCOM*, *3*. https://doi.org/10.1109/INFCOM.2002.1019408

Yick, J., Mukherjee, B., & Ghosal, D. (2008). Wireless sensor network survey.

Computer Networks, *52*(12). https://doi.org/10.1016/j.comnet.2008.04.002

Yu, H., & Kim, Y. (2020). New RSA encryption mechanism using one-time encryption keys and unpredictable bio-signal for wireless communication devices. *Electronics (Switzerland)*, *9*(2). https://doi.org/10.3390/electronics9020246

Yu, Y., Li, K., Zhou, W., & Li, P. (2012). Trust mechanisms in wireless sensor networks: Attack analysis and countermeasures. *Journal of Network and Computer Applications*, *35*(3). https://doi.org/10.1016/j.jnca.2011.03.005

Zhou, Y., Fang, Y., & Zhang, Y. (2008). Securing wireless sensor networks: A survey. *IEEE Communications Surveys and Tutorials*, *10*(3). https://doi.org/10.1109/COMST.2008.4625802