IoT-NUMS: Evaluating NUMS Elliptic Curve Cryptography for IoT Platforms

Zhe Liu, *Senior Member, IEEE*, and Hwajeong Seo, *Member, IEEE*

***Abstract*—In 2015, NIST held a workshop calling for new candidates for the next generation of elliptic curves to replace the almost two-decade old NIST curves. Nothing Upon My Sleeves (NUMS) curves are among the potential candidates presented in the workshop. Here, we present the first implementation of the NUMS256, NUMS379, and NUMS384 curves on two types of embedded devices. The implementations, which exhibit regular, constant-time execution to protect against timing and simple side-channel attacks, set new speed records and advance the state-of-the-art of curve-based (without endomorphism) scalar multiplication on 8-bit AVR and 32-bit ARM11 microcontrollers. For example, our NUMS256 implementation computes a scalar multiplication in ∼1.4 million cycles on a low-power 32-bit ARM11 microcontroller using mixed C and assembly language. These results demonstrate the potential of deploying IoT-NUMS on constrained and low-power applications such as protocols for the Internet of Things.**

***Index Terms*—NUMS curves, AVR ATmega, ARM11, efficient software implementation.**

I. Introduction

NEW security issues and designs for the Internet of Things (IoT) have been an important area of research due to the emergence of IoT applications. In general, the use cases include resource constrained embedded processors, which introduce challenges on the use of Public Key Cryptography (PKC) due to the lack of available memory and higher costs attached to energy consumption. PKC protocols including digital signatures and key agreement schemes usually lead to significant overheads in terms of execution time and energy consumption, which are undesirable for the low-end battery-powered IoT processors. A common sensor node features an 8-bit microcontroller clocked at a frequency of less than 32 MHz and equipped with a few kilobytes (KB) of data and stack memories and up to 256 KB of flash memory for storing program code and constants. Under these computational constraints, the most precious resource of IoT device is energy. Once deployed in the field, the platforms are expected to work several months or years, with the limited energy supplied by two AA batteries that cannot be easily replaced or even recharged. To minimize the energy consumption, lightweight PKC implementations are a fundamental requirement. Among the existing candidates for PKC, elliptic curve cryptography (ECC) is the most promising one compared to other finite field-based (Diffie-Hellman) or integer-factoring-based cryptosystems (RSA). While the best classical attacks against RSA and Diffie-Hellman over finite fields are sub-exponential attacks while the Elliptic Curve Discrete Logarithm Problem (ECDLP) remains exponential and are based on the Pollard's rho attack. This translates into smaller key sizes and bandwidth occupation for ECC. The features of ECC has also promote the public-key cryptography application in IoT solutions as, in many cases, transmitting a bit is about orders of magnitude more expensive than processing a bit in terms of energy consumption in Wireless Sensor Networks for example.

Many elliptic curves are already standardized for several years, for example, NIST standardized 15 elliptic curves, designed by NSA, in the FIPS 186-2 standard together with the ECDSA digital signature scheme in the year of 2000. On the other hand, Edward Snowden, who worked many years for NSA revealed in 2013 that the Dual_EC_DRBG algorithm, presented as a cryptographically secure pseudo-random number generator (CSPRNG), would possibly contain a backdoor. In addition to that, new elliptic curves models and optimized parameters with protection against side-channel attacks have been found in the last years and therefore there is consensus in the cryptographic community that elliptic curve standards should be upgraded.

The first really efficient ECC software for an 8-bit processor was introduced by Gura *et al.* [4] at CHES 2004. They reported an executed time of only $6.48 \cdot 10^6$ clock cycles for a full scalar multiplication over a 160-bit SECG-compliant prime field on the ATmega128. This high speed is mainly due to a smart optimization of the multiple-precision multiplication, the nowadays widely used *hybrid method*. Since the publication of Gura et al's paper, numerous research has been devoted to improve the efficiency of ECC software implementation on the ATmega128, overwhelmingly concentrating on enhancing the hybrid multiplication technique or developing more efficient variants of it. For instance, a further improvement of the hybrid method was reported by Scott and Szczechowiak [11]

Manuscript received December 29, 2017; revised April 19, 2018; accepted June 15, 2018. Date of publication July 13, 2018; date of current version September 13, 2018. This work was supported in part by the National Natural Science Foundation of China under Grant 61802180, in part by the Natural Science Foundation of Jiangsu Province of China under Grant SBK2018043466, and in part by the Fundamental Research Funds for the Central Universities under Grant NE2018106. The work of H. Seo was supported by the Institute for Information & Communications Technology Promotion Grant through the Korean Government (MSIT), Research on Blockchain Security Technology for IoT Services, under Grant 2018-0-00264. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Damien Vergnaud. *(Corresponding author: Zhe Liu.)*

Z. Liu is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China (e-mail: sduliuzhe@gmail.com).

H. Seo is with Hansung University, Seoul 02876, South Korea.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2018.2856123

via utilizing so-called "carry catcher" registers to reduce the cycle count to 2651. Uhsadel *et al.* [24] proposed an optimized method based on the hybrid method and managed to decrease the execution time to 2881 cycles for a (160×160)-bit multiplication (without modular reduction). The so-called *operand-caching method* which was presented by Hutter and Wenger [6] is the currently one of the fastest approaches of the multi-precision multiplication of two large integers on the ATmega128, following a similar idea as the hybrid technique, namely exploiting the large number of general-purpose registers to store (parts) of the operands [6]. This technique significantly reduced the number of required load instructions via caching of operands outperforming the related works by a factor of 10% to 23%. Seo and Kim [4] improved this method minimizing the number of needed load/store instructions and managed to boost the speed of total multiplication for public key cryptography. Moreover, there exist several ECC implementations on ARM processors in the published literature. The first to implement ECC over Cortex-M0+ were Wenger *et al.* [1] in 2013, who presented a side-channel protected and resource saving scalar multi-precision multiplication algorithm with constant running time. Later, they revisited their work and improved the performance by harnessing the drop-in module as well as the advanced MAC hardware extension in CHES 2014 [25]. In 2015, Düll *et al.* [7] proposed several optimized implementation approaches of Curve25519 across a wide range of processor types including 8-bit AVR ATmega, 16-bit MSP430 and 32-bit ARM Cotex-M0, setting new speed records for scalar-multiplication in constant runtime. Furthermore, Longa [23] presented a high-speed implementation of the recently proposed elliptic curve Four $\mathbb{Q}$ [21] over 32-bit ARM processors with high security. Later in 2018, Liu *et al.* [15] further developed this method and managed to set new speed records for curve-based scalar multiplication with constant time on 8/16/32-bit IoT embedded devices.

In 2014, Microsoft Research team introduced NUMS curves, a set of high-security elliptic curves designed to employ the state-of-the-art algorithms with respect to speed and side-channel countermeasures, which cover three security levels (128-, 192- and 256-bit security) [2], [13], [14]. The curves have a very simple and deterministic generation procedure so that the possibility of introducing backdoors is greatly minimized. For instance, in the previous curve generation procedure by NIST/NSA, a seed would be chosen "at random" and its hash would give rise to the curve coefficients. In this procedure, there is no way to check if the parameter designer had tested a large amount of distinct seeds until they had found a curve with some weaknesses or backdoors. In particular, the new approach for NUMS does not allow for such a flexibility in choosing the curve coefficients as it is explained in Section II. The underlying NUMS finite fields are also chosen in a way that arithmetic is fast while prioritizing security at the same time.

NUMS curves were previously implemented on some high-performance end platforms including x64 devices running Windows or Linux and achieved high-performance on those target platforms [14]. The implementation results show that NUMS curves are promising candidate for new elliptic curve standards. On the other hand, due to the fact that NUMS is a relatively recent proposal, there exist no implementations on low-power embedded devices in the previous work.

In this paper, we first focus on the low-level algorithms for typical NUMS finite field arithmetic targeting some typical candidates on IoT platforms, which include 8-bit AVR and 32-bit ARM11 processors. Then, we provide carefully tailored assembly implementations for multi-precision modular multiplication and squaring operations for the finite fields defined in the three NUMS curves. In particular, we implement NUMS curves including NUMS256, Ted37919, and NUMS384, which combine the individual computational advantages of the twisted Edward and Montgomery curves and efficient pseudo-Mersenne primes. We achieve record-setting execution times for scalar multiplication over 256, 379, and 384-bit security prime fields. For example, NUMS256 curve only requires 1.357 M cycles on 32-bit ARM11 processor, which is more than1.6x faster than the widely-used Curve25519. Our experimental results also demonstrate that NUMS could also achieve good performance on resource-limited IoT devices.

The remainder of this paper is organized as follows. In Section II, we recap the NUMS curves and related choice parameters. In Section III, we discuss the 8-bit AVR architecture and its features. In Section IV, we introduce efficient multiplication and squaring for implementing NUMS256, Ted37919, and NUMS384 on 8-bit AVR and 32-bit ARM11 microcontrollers. In Section V, we implement NUMS256, Ted379 and NUMS384 by using a mixed of C and assembly languages. Finally, Section VI concludes the paper.

II. Preliminaries: NUMS Curves

The 15 NIST-standardized curves in FIPS 186-2 do not include all the advances in elliptic curve cryptography and side-channel resistance anymore and new replacement curves are being suggested by the cryptographic community in the last few years.[1] Furthermore, industry companies have a priority demand for cipher-suites including perfect forward secrecy support. With all of these new requirements in mind, high-secure and fast arithmetic elliptic curves, namely NUMS curves, were proposed [14] including important characteristics. First, the NUMS curves support standard security levels, including 128-bit, 192-bit, and 256-bit. Second, they use a rigid technique for parameter generation where given the security level as an input parameter, the curve and finite field parameters are deterministically computed and therefore the attack surface for backdoored constants and parameters is reduced. Also, the underlying primes are defined to be close to a power of 2 integer so that fast modular reduction techniques can be employed such as the Barrett reduction [8]. Third, the NUMS curves work with the existing ECC protocol infrastructure such as TLS 1.2, X.509v3/PKIX, and CMS. Fourth, the NUMS curves can also achieve good performance on both key agreement and digital signature schemes given

[1]https://safecurves.cr.yp.to/

that they can be defined over the Weierstrass and the Twisted Edwards forms. Fifth, the NUMS curves support standard elliptic curve point representations compatible with existing (x, y) coordinate encoding formats. Finally, NUMS offer an implementation-friendly standard group and field order bit lengths, allowing for a typical 64-bit length alignment (i.e., modern CPU register boundary).

In this paper, we target three NUMS curves covering 128-bit, 189-bit, and 192-bit security levels. NUMS256 and NUMS384 are defined over Montgomery curve and Ted37919 is defined over Twisted Edward curve, respectively. The detailed descriptions are as follows.

To generate Weierstrass NUMS-based curves, we are given security level $s \in \{128, 192, 256\}$, and the prime p is selected as a pseudo-Mersenne prime and parameterized by $p = 2^{2s} - c$ for a positive integer c. The prime will be chosen to be the one for the smallest c such that $p = 2^{2s} - c$ is prime. For the three values of s above, the resulting primes satisfy $p \equiv 3 \bmod 4$. The Weierstrass model of the respective curves will be such that for the equation

$$E_b/\mathbb{F}_p : y^2 = x^3 - 3x + b.$$

We select the coefficient $b \neq \pm 2$ so that it has the smallest possible absolute value and both the order of $|E_b|$ and its quadratic twist $|E'_b|$ are both prime.

Later, NUMS were extended to primes satisfying $p \equiv 1 \bmod 4$ in order to enable the use of the fast complete formulas in the twisted Edwards model. They are curves of the form

$$E_d/\mathbb{F}_p : -x^2 + y^2 = 1 + dx^2y^2,$$

whose quadratic twist is automatically given by

$$E'_d/\mathbb{F}_p : -x^2 + y^2 = 1 + (1/d)x^2y^2.$$

The criteria to select a curve of this form consists of picking a prime which is close to a power of 2 and has a small constant c. To find the curve equation coefficient, look for the smallest d (in absolute value) which is a non-square in $\mathbb{F}_p$ such that $|E_d| = h \cdot r$ and $|E'_d| = h' \cdot r'$ where r, r' are primes and $h = h' = 4$.

A. NUMS256 (or Numsp256D1)

This is a Weierstrass curve offering a security of about 127.8 bits against Pollard's rho attack. The underlying prime is given by $s = 128$ and $c = 189$, i.e.,

$$p_{256} = 2^{256} - 189.$$

The Weierstrass form uses the coefficient $b = 152961$ and is given by

$$E_{152961} : y^2 = x^3 - 3x + 152961,$$

while its isomorphic Montgomery form is given by the following

$$E_{M,-61370} : y^2 = x^3 - 61370x^2 + x.$$

B. Ted37919 (or Numsp379T1)

This is a twisted Edwards curve offering security of about 188 bits. The selected prime is

$$p_{379} = 2^{379} - 19.$$

The respective twisted Edwards equation sets $d = 143305$ and is given by

$$E_{143305} : -x^2 + y^2 = 1 + 143305x^2y^2.$$

This curve also has an isomorphic Montgomery equivalent with smallest constant $(A + 2)/4$ which allows for more efficient scalar multiplication.

C. NUMS384 (or Numsp384D1)

This is a Weierstrass curve offering security of about 192 bits. The underlying prime sets $s = 192$ and $c = 317$, i.e.,

$$p_{384} = 2^{384} - 317.$$

Its Weierstrass equation uses $b = 34568$ and is given by

$$E_{34568} : y^2 = x^3 - 3x + 34568.$$

In the following, we also list other three well-known efficient elliptic curves that are quite related to our work.

D. Four$\mathbb{Q}$

Four$\mathbb{Q}$, introduced by Costello and Longa in 2015 [21], is defined by the complete twisted Edwards [22] equation:

$$\mathcal{E}/\mathbb{F}_{p^2} : -x^2 + y^2 = 1 + dx^2y^2,$$

where the quadratic extension field $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ for $i^2 = -1$ and $p = 2^{127} - 1$, and $d = 125317048443780598345676279555970305165 \cdot i + 4205857648805777768770$. The prime order subgroup $\mathcal{E}(\mathbb{F}_{p^2})[N]$, where N is the 246-bit prime corresponding to $\#\mathcal{E}(\mathbb{F}_{p^2}) = 392 \cdot N$, is used to carry out cryptographic computations. In this subgroup, the neutral element is given by $\mathcal{O}_{\mathcal{E}} = (0, 1)$ and the inverse of a point (x, y) is given by $(-x, y)$. Four$\mathbb{Q}$ provides two different types of endomorphisms(i.e. the CM and Frobenius endomorphisms), which ensures its fastest implementations for key-exchange and signature scheme.

E. Curve25519

Curve25519 was proposed by Daniel J. Bernstein, it provides roughly 128-bit security level together with high performance on modern processors [17]. The arithmetic is with equation

$$\mathcal{E}/\mathbb{F}_p : y^2 = x^3 + 486662\, x^2 + x$$

defined over the field $\mathbb{F}_{2^{255}-19}$.

F. Curve448

Curve448 was suggested by Hamburg [19] to provide the high-end security together with the flexible implementation for a wide range of different software platforms. In particular, the curve satisfies the SafeCurves policies and eliminates the security vulnerabilities and flaws in previous NIST curves [19]. The arithmetic is with equation.

$$\mathcal{E}/\mathbb{F}_p : y^2 + x^2 = 1 + dx^2y^2$$

defined over the field $\mathbb{F}_{2^{448}-2^{224}-1}$ and $d = -39081$.

III. Implementation on 8-bit AVR Microcontroller

A. 8-bit AVR Architecture

Today, many widely-used low-cost smartcards and wireless sensor nodes are equipped with 8-bit AVR microcontrollers (e.g. the MicaZ mote). AVR microcontroller (such as the Atmel ATmega128 or the ATxmega256A3) has an 8-bit RISC instruction set and a modified Harvard architecture that features 32 8-bit general-purpose registers denoted by `R0,...,R31`. From this pool of registers, the last three pairs, called X (`R27:R26`), Y (`R29:R28`) and Z (`R31:R30`), are used as 16-bit address pointers to load and store data from memory. The AVR instruction set supports a total of 133 instructions, and each instruction has a fixed latency. For example, the ordinary arithmetic/logical instructions such as addition (`ADD`) and addition with carry (`ADC`), are executed in a single clock cycle, while the unsigned multiplication (`MUL`) and load/store instructions cost two clock cycles.

Depending on the concrete type of the processor, AVR microcontrollers are different with the memory capacity. For example, the ATxmega256A3 has 256KB of programmable flash memory, 16KB of SRAM, and 4KB of EEPROM. There exist numerous development tools for AVR. For our actual benchmarks we adopt the ATxmega256A3 which operates at a maximum frequency of 32MHz, we run the code on AVR Studio, which features an assembler and a cycle-accurate graphical simulator.

The cycle count measurement methodology consists of monitoring the number of cycles for each operation (including function call overheads) using the simulator graphical interface cycle monitor. In addition, because our scalar multiplication algorithms are designed to be constant-time, every execution of scalar multiplication will take the same number of cycles independently of the input scalar and point coordinates. We confirm that by monitoring a single execution *versus* a 1000$\times$ average execution. Most of the software implementation on AVR microcontrollers is written in both mixed C and Assembly code. The C function-call ABI specifies that the first three 16-bit arguments (e.g., pointers) are passed in register pairs (`R24:R25`), (`R22:R23`), and (`R21:R20`). Furthermore, it specifies that registers `R2-R17`, `R28`, and `R29` are "called-saved" registers, and the register `R1` is assumed by the compiler to always contain zero thus has to be set to zero before returning from a C function.

B. Multiplication and Squaring

Finite field multiplication and squaring are the most expensive low-level operations, which contributes to more than 70% of the whole ECC scalar multiplication. In the following, we will describe our implementation for the NUMS primes.

Multiplication: A simple and easy-to-implement technique for multi-precision multiplication is the operand scanning method, also known as schoolbook method. An alternative way of performing multi-precision multiplication is the product scanning method, which was the first to describe an efficient implementation on an Intel processor but later found to be efficient on embedded devices as well. In 2011, Hutter and Wenger presented a very interesting idea to increase the performance of multiplication by sophisticated caching of operands [6]. Their method significantly reduces the number of needed memory-access instructions(e.g., `LD`, `ST`). By following Hutter-Wenger's work, Seo and Kim proposed the consecutive operand-caching method to further reduce the number of addition with carry instruction [4]. Another interesting work based on product scanning method belongs to Liu et al., who improved Comba's result for small devices using the so-called Reverse Product-Scanning approach [12]. The Karatsuba algorithm is a fast multiplication algorithm which reduces the time complexity of multi-precision multiplication down to $O(n^{log_2 3})$ asymptotically for an n-digit representation. In this work, given the 64-bit alignment of the NUMS primes, we take advantage of the Karatsuba multiplication in multiple layers. Also, due to a limited number of general purpose registers, we perform the modular multiplication and squaring operations separately, namely, the reduction is executed independently of the computation of the multi-precision multiplication and squaring and therefore. The detailed description is as follows.

1) 256-bit Multiplication and Squaring: We employ the state-of-the-art multiplication and squaring for 256-bit in our implementation, namely the subtractive Karatsuba implementation proposed by Hutter and Schwabe [9] and the squaring in [10]. Our 256-bit multi-precision multiplication performs three times of subtractive Karatsuba multiplications, which enhances the performance by a factor of 18.8% when compared to consecutive operand caching. For the squaring operation, the hybrid Karatsuba method [10] combines both advantages of Karatsuba method and sliding block doubling, which improves the performance by 7.2% compared to sliding block doubling. Both method represents the state-of-the-art implementation for multi-precision multiplication and squaring on 8-bit AVR processors. For reduction, we also implement it in assembly language.

2) 379/384-bit Multiplication and Squaring: The 384-bit multiplication/squaring on AVR processor is not studied in previous works [9], [10]. In order to construct the 384-bit cases, we employ Hutter-Schwabe's Karatsuba implementation [9] to perform the 192-bit multiplication, while the squaring is realized by the optimized Karatsuba squaring [10]. That is to say, we actually adopt three-level Karatsuba method for the 384-bit multiplication and squaring, which are implemented in pure Assembly. The multiplication requires

Algorithm 1 Constant-Time Subtractive Karatsuba Multiplication (Subtractive Fashion)

```
Require: Two 384-bit operand $A = AH \cdot 2^{192} + AL$ and $B = BH \cdot 2^{192} + BL$.
Ensure: The product $Z = A \cdot B$.
1: Compute abstract value of $ADIFF = |AH - AL|$.                    {int_neg; int_sub}
2: Compute abstract value of $BDIFF = |BH - BL|$.                    {int_neg; int_sub}
3: Compute abstract value of $ZDIFF = |ADIFF \cdot BDIFF|$.          {int_neg; karatsuba_mul_192 [9]}
4: Compute $ZL = AL \cdot BL$.                                       {karatsuba_mul_192 [9]}
5: Compute $ZH = AH \cdot BH$.                                       {karatsuba_mul_192 [9]}
6: Compute $Z = ZH \cdot 2^{384} + (ZL + ZH - ZDIFF) \cdot 2^{192} + ZL$.   {Three int_add; int_add_word}
7: return $Z$
```

Algorithm 2 Constant-Time Subtractive Karatsuba Squaring (Subtractive Fashion)

```
Require: Two 384-bit operand $A = AH \cdot 2^{192} + AL$.
Ensure: The product $Z = A^2$.
1: Compute abstract value of $ADIFF = |AH - AL|$.                    {int_neg; int_sub}
2: Compute abstract value of $ZDIFF = ADIFF^2$.                      {karatsuba_sqr_192 [10]}
3: Compute $ZL = AL^2$.                                              {karatsuba_sqr_192 [10]}
4: Compute $ZH = AH^2$.                                              {karatsuba_sqr_192 [10]}
5: Compute $Z = ZH \cdot 2^{384} + (ZL + ZH - ZDIFF) \cdot 2^{192} + ZL$.   {Two int_add; int_sub, int_add_word}
6: return $Z$
```

10,633 clock cycles and consumes 15,502 bytes while the 384-bit squaring needs 7,278 clock cycles and 10,376 bytes. The detailed 384-bit multiplication and squaring are available in Algorithm 1 and Algorithm 2, respectively. We use constant-time subtractive Karatsuba method to perform a 384-bit multiplication/squaring with three 192-bit multiplications/squaring operations.

3) Memory-Optimized Multiplication and Squaring: A low-end 8-bit AVR processor provides limited program ROM typically ranging from 64-256 KB. If the application is highly code memory consuming, then little ROM would be left available for cryptographic purposes. In order to strike the balance between code size and performance, we introduce a Karatsuba-Reverse-Product-scanning (KRPS) method. This memory efficient technique supports practically fast computation within 2 KB of memory. For the 384-bit case, the KRPS is 26% slower in performance but saves about 11% of the code size; the RPS squaring is 14% slower in performance but requires only 8% of the code size. A concrete comparison is given in Table I.

4) Fast Incomplete Reduction for $2^{379} - 19$: The reduction equation can be realized as $2^{379} \equiv 19 \bmod p_1$ by following the definition of the modulo. A straightforward reduction is to repeatedly replace $ZH \cdot 2^{379}$ with $ZH \cdot 19$ where ZH consists of the bits if the argument greater than the 379th one. However, since 379 is not an 8-bit friendly number (i.e. a multiple of 8), this method requires a number of shifting operations on intermediate results. In order to optimize the shift operation, we introduce a fast modular reduction technique for the NUMS prime $2^{379} - 19$. The reduction is inspired by the observation that

$$2^{379} \equiv 19 \Longrightarrow 2^{384} \equiv 608 \bmod p_1$$

The main idea is to perform the first round reduction with $2^{384} \equiv 608 \bmod p_1$ and then do the second round reduction with $2^{379} \equiv 19 \bmod p_1$. Instead of doing a complete reduction with the final result in the range of $[0, 2^{379} - 19]$, the reduction is carried out in an incomplete way which allows for the reduction result to be kept in the range of $[0, 2^{380}]$. The fast reduction algorithm shown in Algorithm 3 can be described as follows:

1) We perform the first reduction (line 1), i.e. the computation of $(c', Z') = ZH \cdot c_1 + ZL$. Instead of computing the whole $ZH \cdot c + ZL$, we only compute $T[0 \sim 5] = ZH[44 \sim 47] \cdot c + ZL[44 \sim 47]$, which essentially determine the part which is over 384-bit.
2) After addition, the sum is longer than 384-bit. We separated the sum into two parts. The first part is lower

TABLE I

EXECUTION TIME AND CODE SIZE OF 384-bit MULTIPLICATION AND SQUARING WITH DIFFERENT COMBINATIONS

Integer	Combination	Time (cycles)	ROM (bytes)	Karatsuba
384-bit MUL	(1) + (3)	11,898	5,474	3 levels
384-bit SQR	(2) + (4)	8,037	3,800	3 levels
384-bit MUL	(1) + (5)	14,382	1,704	1 level
384-bit SQR	(6)	8,505	832	No
384-bit MUL	(7)	10,633	15,502	3 levels
384-bit SQR	(8)	7,278	10,786	3 levels

(1): Constant-time subtractive Karatsuba multiplication (in ANSI C).

(2): Constant-time subtractive Karatsuba squaring (in ANSI C).

(3): 192-bit Karatsuba multiplication [10] (in Asembly).

(4): 192-bit Karatsuba squaring [11] (in Assembly).

(5): Optimized RPS multiplication (in Assembly).

(6): Optimized RPS squaring (in Assembly).

(7): 384-bit Karatsuba multiplication (in Assembly).

(8): 384-bit Karatsuba Squaring (in Assembly).

Algorithm 3 Fast (Incomplete) Reduction for p_{379}

```
Require: A 2n-bit product $Z = ZH \cdot 2^{384} + ZL$, the constants $c1 = 2^5 \cdot 19$ and $c2 = 19$.
Ensure: The incomplete reduction result $R = Z \bmod p_1 \in [0, 2^{380}]$.
1: $T1[0 \sim 5] \leftarrow ZH[44 \sim 47] \times c1 + ZL[44 \sim 47]$    {First reduction: $2^{384} \equiv 608 \bmod p_1$}
2: $ZL[44 \sim 47] \leftarrow ((T1[3]\&\texttt{0x07}) \parallel T1[0 \sim 2])$
3: $T2[0 \sim 2] \leftarrow (T1[3 \sim 5] \gg 3) \times 19$    {Second reduction: $2^{379} \equiv 19 \bmod p_1$}.
4: $R[0 \sim 47] \leftarrow (ZH[0 \sim 43] \times c1) + ZL[0 \sim 47] + T2[0 \sim 2]$
5: return $R$
```

Algorithm 4 Fast (Incomplete) Reduction for p_{384}

```
Require: A 2n-bit product $Z = ZH \cdot 2^{384} + ZL$, the constants $c = 317$.
Ensure: The incomplete reduction result $R = Z \bmod 2^{384} - 317 \in [0, 2^{384}]$.
1: $T1[0 \sim 5] \leftarrow ZH[44 \sim 47] \times c + ZL[44 \sim 47]$
2: $ZL[44 \sim 47] \leftarrow T1[0 \sim 3]$
3: $T2[0 \sim 3] \leftarrow T1[4 \sim 5] \times c$
4: $\{carry, R[0 \sim 47]\} \leftarrow (ZH[0 \sim 43] \times c) + ZL[0 \sim 47] + T2[0 \sim 3]$
5: $R[0 \sim 47] \leftarrow carry \times c + R[0 \sim 47]$
6: return $R$
```

than 2^{379}, namely, $((T1[3]\&\texttt{0x07}) \parallel T1[0 \sim 2])$, we directly store them in memory (line 2).

3) The second part is higher than 2^{379}, namely, $(T1[3 \sim 5]) \gg 3$. we multiply it by $c2$, and store the product in three temporary registers $T2[0 \sim 2]$.
4) Finally, the remaining parts ($ZH[0 \sim 43]$) are multiplied by constant c and added to the intermediate results ($ZL[0 \sim 47]$) and reduction results $T2[0 \sim 2]$.

Algorithm 3 requires an execution time of 1018 clock cycles including the stack operation at the beginning and end of the Assembly function (e.g., PUSHs and POPs).

5) Fast Incomplete Reduction for $2^{384} - 317$: Similar as Algorithm 3 for $2^{379} - 19$, the reduction of p_{384} needs an execution time of 1157 clock cycles. The fast reduction algorithm shown in Algorithm 4 can be described as follows:

1) We perform the first reduction (line 1), i.e. the computation of $(c', Z') = ZH \cdot c + ZL$. Instead of computing the whole $ZH \cdot c + ZL$, we only compute $T[0 \sim 5] = ZH[44 \sim 47] \cdot c + ZL[44 \sim 47]$, which essentially determines the part which is over 384-bit.
2) We separate the sum into two parts. The first part is lower than 2^{384}, namely, $(T1[0 \sim 3])$. Then, we directly store them in memory (line 2).
3) The second part is higher than 2^{384}, namely, $T1[4 \sim 5]$. We multiply it by c, and store the product in three temporary registers $T2[0 \sim 3]$.
4) The remaining parts ($ZH[0 \sim 43]$) are multiplied by constant c and added to the intermediate results ($ZL[0 \sim 47]$) and reduction results $T2[0 \sim 3]$.
5) Finally, the *carry* bit is multiplied by constant c and added to results $R[0 \sim 47]$.

6) Field Inversion: Constant-time inversion can be obtained by using Fermat's little theorem $a^{-1} = a^{p-2} \bmod p$. The inversion of NUMS256 requires $255S + 12M$ as shown in Algorithm 5, which costs 1, 218, 645 cycles. Similarly, the inversion operation of Ted379 requires an execution time of $378S + 12M$ as shown in Algorithm 6. On an 8-bit AVR processor, the inversion requires 3, 304, 148 clock cycles. For NUMS384, the Fermat-based inversion requires $383S + 13M$. The addition chain is given in Algorithm 7.

Algorithm 5 Fermat-Based Inversion Mod p_{256}

```
Require: Integer $A$ satisfying $1 \leq A \leq p - 1$.
Ensure: Inverse $Z = A^{p-2} \bmod p = A^{-1} \bmod p$.
1: $a_6 \leftarrow (a^2 \cdot a)^2$    {exp: 6, cost: 2S+1M}
2: $t_1 \leftarrow ((a_6)^2)^2 \cdot a_6 \cdot a$    {exp: $2^5 - 1$, cost: 2S+2M}
3: $t_2 \leftarrow ((t_1)^2)^5 \cdot t_1$    {exp: $2^{10} - 1$, cost: 5S+1M}
4: $t_3 \leftarrow ((a_6)^2)^2 \cdot a_6 \cdot a$    {exp: $2^{20} - 1$, cost: 10S+1M}
5: $t_4 \leftarrow (((t_3)^2)^{10} \cdot t_2)^2 \cdot a$    {exp: $2^{31} - 1$, cost: 11S+2M}
6: $t_5 \leftarrow (((t_4)^2)^{31}) \cdot t_4$    {exp: $2^{62} - 1$, cost: 31S+1M}
7: $t_6 \leftarrow (((t_5)^2)^{62}) \cdot t_5$    {exp: $2^{124} - 1$, cost: 62S+1M}
8: $t_7 \leftarrow (((t_6)^2)^{124}) \cdot t_6$    {exp: $2^{248} - 1$, cost: 124S+1M}
9: $Z \leftarrow (((((t_6)^2)^2 \cdot a)^2)^6) \cdot a$    {exp: $2^{256} - 191$, cost: 8S+2M}
10: return $Z$
```

C. Summary of Execution Time for Field Arithmetic Operations

The execution time of field arithmetic of NUMS256, Ted379 and NUMS384 is summarized in Table II. The NUMS256 shows relatively higher performance than others since NUMS256 is performed over short integer. Interestingly, TED379 shows the fastest finite field addition since this curve can avoid reduction by taking advantages of incomplete representation. TED379 shows higher performance than NUMS384 in other operations as well. Overall, TED379 achieves the balanced results between security and performance.

TABLE II

EXECUTION TIME OF FIELD ARITHMETIC FOR NUMS256, TED37919 AND NUMS384 ON AVR MICROCONTROLLER (U: UNROLL, P: PARTLY UNROLL AND L: LOOPED FASHIONS), CMUL: CONSTANT MULTIPLICATION WHERE NUMS256 IS 16·256-bit ($0x3BEF$) TED379 IS 24·379-bit ($0x22FCA$) AND NUMS384 IS 16 · 256-bit ($0x2D25$)

Operation	ADD	SUB	MUL	SQR	INV	CMUL
p_{256}	550	550	6,301	4,489	1,218,645	830
p_{379} (U/P/L)	363	686	11,721/12,971/15,455	8,363/9,081/10,496	3,304,148 /3,566,477/4,132,598	1424
p_{384} (U/P/L)	959	959	11,862/13,113/15,590	8,501/9,254/10,663	3,411,204/3,715,866/4,287,720	1227

Algorithm 6 Fermat-Based Inversion Mod p_{379}

```
Require: Integer $A$ satisfying $1 \le A \le p-1$.
Ensure: Inverse $Z = A^{p-2} \bmod p = A^{-1} \bmod p$.
1: $a_2 \leftarrow a^2$                                   {exp: 2, cost: 1S+0M}
2: $a_9 \leftarrow (a_2)^{2^2} \cdot a$                    {exp: 9, cost: 2S+1M}
3: $a_{11} \leftarrow (a_9) \cdot a_2$                     {exp: 11, cost: 0S+1M}
4: $t_1 \leftarrow (a_{11})^2 \cdot a_9$                   {exp: $2^5-1$, cost: 1S+1M}
5: $t_2 \leftarrow (t_1)^{2^5} \cdot t_1$                  {exp: $2^{10}-1$, cost: 5S+1M}
6: $t_3 \leftarrow (t_2)^{2^1} \cdot a$                    {exp: $2^{11}-1$, cost: 1S+1M}
7: $t_4 \leftarrow (t_3)^{2^{11}} \cdot t_3$               {exp: $2^{22}-1$, cost: 11S+1M}
8: $t_5 \leftarrow (t_4)^{2^{22}} \cdot t_4$               {exp: $2^{44}-1$, cost: 22S+1M}
9: $t_6 \leftarrow (t_5)^{2^{44}} \cdot t_5$               {exp: $2^{88}-1$, cost: 44S+1M}
10: $t_7 \leftarrow (t_6)^{2^{88}} \cdot t_6$              {exp: $2^{176}-1$, cost: 88S+1M}
11: $t_8 \leftarrow (t_7)^{2^{176}} \cdot t_7$             {exp: $2^{352}-1$, cost: 176S+1M}
12: $t_9 \leftarrow (t_8)^{2^{22}} \cdot t_4$              {exp: $2^{374}-1$, cost: 22S+1M}
13: $Z \leftarrow (t_9)^{2^5} \cdot a_{11}$                {exp: $2^{379}-21$, cost: 5S+1M}
14: return $Z$
```

Algorithm 7 Fermat-Based Inversion Mod p_{384}

```
Require: Integer $A$ satisfying $1 \le A \le p-1$.
Ensure: Inverse $Z = A^{p-2} \bmod p = A^{-1} \bmod p$.
1: $a_3 \leftarrow a^2 \cdot a$                            {exp: 3, cost: 1S+1M}
2: $a_{15} \leftarrow (a_3)^{2^2} \cdot a_3$               {exp: 15, cost: 2S+1M}
3: $t_0 \leftarrow a_{15}^2 \cdot a$                       {exp: $2^5-1$, cost: 1S+1M}
4: $t_1 \leftarrow t_0^{2^5} \cdot t_0$                    {exp: $2^{10}-1$, cost: 5S+1M}
5: $t_2 \leftarrow (t_1)^{2^5} \cdot t_0$                  {exp: $2^{15}-1$, cost: 5S+1M}
6: $t_3 \leftarrow (t_2)^{2^{15}} \cdot t_2$               {exp: $2^{30}-1$, cost: 15S+1M}
7: $t_4 \leftarrow (t_3)^{2^{30}} \cdot t_3$               {exp: $2^{60}-1$, cost: 30S+1M}
8: $t_5 \leftarrow (t_4)^{2^{60}} \cdot t_4$               {exp: $2^{120}-1$, cost: 60S+1M}
9: $t_6 \leftarrow (t_5)^{2^{120}} \cdot t_5$              {exp: $2^{240}-1$, cost: 120S+1M}
10: $t_7 \leftarrow (t_6)^{2^{120}} \cdot t_5$             {exp: $2^{360}-1$, cost: 120S+1M}
11: $t_8 \leftarrow (t_7)^{2^{15}} \cdot t_2$              {exp: $2^{375}-1$, cost: 15S+1M}
12: $t_9 \leftarrow ((t_8)^{2^3} \cdot a_3)^{2^6}) \cdot a$    {exp: $2^{384}-319$, cost: 9S+2M}
13: return $t_9$
```

IV. IMPLEMENTATION ON 32-bit ARM11 MICROCONTROLLER

A. 32-bit ARM11 Architecture

ARM11 is part of 32-bit RISC ARM processor cores. It supports the ARMv6 instruction set, which comprises 16-bit Thumb, 32-bit Thumb-2, and 32-bit ARM instructions. The ARM11 processor has a 8-stage pipeline with unaligned and mixed-endian data access and 16 32-bit registers (`r0:r15`). Instructions that are relevant for our implementation include 32-bit arithmetic and logical instructions.

1) Parameter Passing: for example, function add(a, b, c, d), the address of array a, b, c, d will be stored in registers $R0$, $R1$, $R2$ as well $R3$, respectively, in sequence.

2) Register Usages: ARM processor has 16 registers, all of them are 32 bits wide. $R13$, $R14$ and $R15$ are special registers. $R13$ (SP) holds the stack pointer. $R14$ (LR) is the link register which holds the callers' return address. $R15$ (PC) holds the program counter. $R0$ is the return value. For example, multi-precision multiplication $(\varepsilon, C) = A + B$, the carry ε has to be moved into register $R0$.

3) Instruction Set: ARM 11 instruction set can be found in http://simplemachines.it/doc/arm_inst.pdf. Some basic instructions used in our implementation:

- `UMULL`: 32-bit multiplication. `UMULL R10, R11, R4, R5` means $R11 \cdot 2^{32} + R10 = R4 \cdot R5$.
- `MOV`: move value. `MOV R5, R4`: move the value in $R4$ to $R5$.
- `LDR, STR`: load/store value from/to memory to/from one register. `LDR R3, [R0, #4]`: load the operand in memory address $R0 + 4$ to register $R3$. `STR R5, [R1, #4]`: store the value in $R5$ to the memory address $R1 + 4$. Pre-indexed; for example `STR r0, [r1, # 12]`. Post-indexed: `STR r0, [r1], # 12`.
- `LDMIA, STMIA`: load/store value from/to memory to/from several registers. `LDMIA R0!, R4-R7`, `STMIA R2!, R6-R7`, the symbol `!` means updating the memory address automatically.
- `ADD/ADC/SUB/SBC`: add/sub value between registers. `ADCS R4, R4, R8`: $(\varepsilon, R4) = R4 + R8 + carry$, and then set ε to "carry" flag. The S following `ADC` is used for set status flag.

Different from AVR processors, the ARM instructions described above only cost one clock cycle.

B. Multiplication and Squaring

1) Comparing Different Multiplication Approaches: There exist several methods that can be used for performing the multi-precision multiplication and squaring on 32-bit ARM processor. The first method is called operand scanning method (i.e. Left side of Figure 1). It keeps the multiplier b_i constant and multiplies it with the entire multiple-precision multiplicand $(a_{n-1}, \ldots, a_1, a_0)$ before moving to the next multiplier b_{i+1}. The operand scanning method has advantage when the processor has sufficient registers for keeping entire operands and necessary computation usage. Namely, (1) $n + 1$ registers

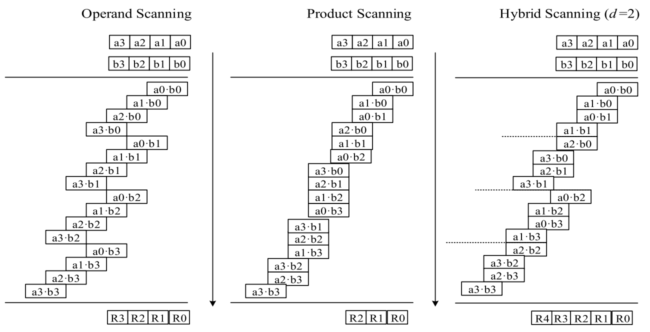

Fig. 1. Operand scanning VS product scanning VS hybrid scanning.

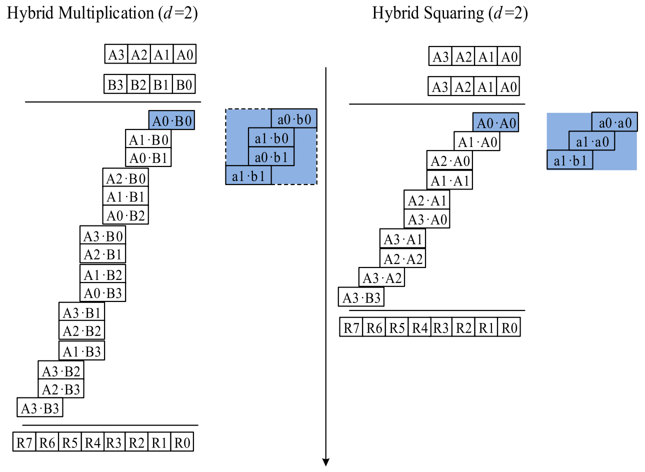

Fig. 2. Hybrid scanning method (d=2) for 256-bit multiplication and squaring.

to keep the operand A and one word of operand B (i.e. b_i); (2) $n+1$ registers for accumulator registers; (3) at least 2 more registers for `UMULL` instruction. In our case, the operand has a length of 256-bit or 384-bit, and the ARM processor has 13 registers available for user, which is, no doubt, not enough for operand scanning method. Another method to perform multi-precision multiplication is called product scanning method, it sums up columns of partial products $a_j \cdot b_i$, where $i + j = l$ for column l. At the end of each column, one 32-bit word (for 32-bit ARM platform) is stored as part of the final multiplication result. This method requires less registers for computation but more load operations (from memory to registers). The third general method for efficient multiplication is Karatsuba method, which replaces the multiplication with addition and subtraction operation. However, Karatsuba method has the advantage only when the cost of multiplication is much more expensive than addition and subtraction, which may not be suitable on 32-bit ARM processor where both of `UMLL` and `ADD` instructions has the same cost, i.e. one clock cycle.

2) Our Choice (Hybrid-Scanning Method): Hybrid scanning method was first proposed by Gura *et al.* [4] in CHES'04 for speeding up the performance of multiplication on AVR platform. Hybrid scanning aims at optimizing for both the number of registers and the number of memory accesses. We employ the product-scanning strategy as the "outer algorithm" and the operand-scanning strategy as the "inner algorithm". That is, hybrid multiplication computes columns that consist of rows of partial products. The savings in memory bandwidth stem from the fact that 32-bit operands of the multiplier are used in several multiplications, but are loaded from memory only once. A comparison of operand scanning, product scanning as well as hybrid scanning methods can be see in [3, p. 8, Table 1].

Our implementation applies this idea for multi-precision multiplication and squaring on ARM platform. For sake of achieving good performance, we allocate the 13 available registers ($R0 - R12$) in the following way:

- $R0$, $R1$, $R2$ for operand address. $mul\_256(A, B, R)$: $R = A \cdot B$.
- $R3$, $R4$ for operand A, $R5$ for operand B.
- $R6$, $R7$, $R8$, $R9$, $R12$ for accumulator $T = R12$, $U = (R6, R7)$, $V = (R8, R9)$
- $R10$, $R11$ for `UMULL` instruction, namely temporary registers for keeping the product.

See Figure 2 for an example of hybrid scanning method for 256-bit multiplication and squaring. Using the IAR Workbench simulator in the context of the ARM11 of the architecture, our implementation of multi-precision multiplication/squaring requires 404/285 and 898/659 cycles for 256-bit and 384-bit, respectively. One may also consider to use the carry-catcher method [11, p. 11] for the implementation of multi-precision multiplication on ARM processor, however, when using one carry-catcher for the function of several carry-catchers, one has to pay extra cost for masking and shifting operations before adding the carry bits to the accumulator registers. The process of masking and shifting is complex and even requires more execution time than straightforward using the hybrid multiplication. For a comparison, Scott and Szczechowiak in [11] reported 580 clock cycles for product scanning method and 487 cycles for carry-catcher method in case of a $192 \cdot 192$ bit multiplication, and our implementation of 256-bit multiplication only costs 543 cycles. Düll *et al.* [7] pointed out that the most efficient strategy for multiplication on ARM Cortex-M0 is Karatsuba method. However, our target platform ARM11 also supports the full word multiplication (i.e. $32 - bit \times 32 - bit \leftarrow 64 - bit$),[2] which ensures hybrid multiplication is slightly more efficient than Karatsuba method for 384-bit length. To confirm this, we evaluated the 256-bit finite field multiplication and 384-bit finite field multiplication with Karatsuba method. Karatsuba method requires execution time of 746 and 1407 clock cycles for 256-bit and 384-bit, respectively, which is slower than the hybrid multiplication implementation (i.e. 543 and 1139 clock cycles for 256-bit and 384-bit, respectively). On the other hand, hybrid multiplication is more efficient in the sense of memory consumption. To summarize, we adopt hybrid multiplication in the ARM11 implementation.

[2]Note that Cortex M0 only provides a half word multiplication (i.e. $32 - bit \times 32 - bit \leftarrow 32 - bit$).

TABLE III

PERFORMANCE COMPARISON OF SCALAR MULTIPLICATION FOR FOUR$\mathbb{Q}$, μKUMMER, NUMS256, TED379, NUMS384, CURVE41417 AND ED448 ON AVR MICROCONTROLLER

Alg.	Four$\mathbb{Q}$ [16]	Kummer [17]	NUMS256	Curve25519 [8]	NIST256 [1]	Ted379	NUMS384	Curve41417	Ed448
Window	–	–	15,807,767	–	34,930,000	42,978,573	44,633,126	49,919,039†	59,421,773†
Ladder $\star$	9,948,900	19,945,200	15,125,232	13,900,397	–	41,931,327	44,051,162	48,727,345†	58,619,275†

†: Estimated results.

*: **Constant multiplication** $(A+2)/4$ **is used in point doubling**.

V. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we summarize our implementation results for 8-bit AVR and 32-bit ARM11 microcontrollers. For our benchmarks, the implementations were compiled and evaluated using IAR Embedded Workbench. Our IoT-NUMS implementations are based on [13] for the case of variable-base scalar multiplication.

A. Implementation Results

1) Results on AVR: The execution time of Four$\mathbb{Q}$, Kummer, NIST256, NUMS256, Ted37919, NUMS384, Curve41417 and Ed448 is given in Table III. We used window method ($w = 5$) for twisted Edwards curve and Montgomery ladder algorithm for Montgomery form. We test the current implementation of Montgomery ladder method for Ted379 and NUMS384, which require 41,931,327 and 44,051,162 clock cycles. We also give the reasonable estimated results for Curve41417 and Ed448 in Table III. For 128-bit security level, Four$\mathbb{Q}$ achieved the highest performance because of the feature of endomorphism. Unfortunately, Four$\mathbb{Q}$ implementation requires roughly 38 KB for code size. When compared to Curve25519 implementation in [7], the NUMS256 implementation is slightly slow, the main reason is that the prime $2^{255} - 19$ provides more efficient reduction operation. However, NUMS curves could support standard security levels, including 128-bit, 192-bit, and 256-bit and have a rigid technique for parameter generation. The implementation of NUMS256 curve only requires 18$\sim$20 KB with reasonably fast performance. For the higher security level, NUMS384 and Ted37919 curves show a very good performance as well.

2) Results on ARM11: The execution time of field arithmetic of NUMS256, Ted37919 and NUMS384 is summarized in Table IV. The finite field multiplication and squaring operations require 543/428, 1126/884, and 1139/898 clock cycles for NUMS256, Ted37919, and NUMS384 curves. For the finite field addition, Ted37919 shows the highest performance due to the efficient incomplete reduction method.

Including the function call, the point doubling/addition of NUMS256 costs 3659/5285 clock cycles. For NUMS384, the point doubling/addition requires an execution time of 7475/10801 clock cycles, respectively. For the scalar multiplication, we tested both twisted Edward curve and Montgomery curve. The detailed results are given in Table V. Compared to the C implementation of Curve25519 on ARM11 by running SUPERCOP, the NUMS256 implemented in mixed C-assembly achieves a speed up of 1.64X. When compared to Goldilock, the NUMS256, Ted37919 and NUMS384 outperform by a factor of 8.2, 2.8 and 2.7, respectively. The comparison graph is drawn in Figure 3. The figure shows that IoT-NUMS successfully fills the medium security level and achieves the highest performance among them.

TABLE IV

EXECUTION TIME OF FIELD ARITHMETIC FOR NUMS256, TED37919 AND NUMS384 ON ARM MICROCONTROLLER (UNROLL AND LOOPED FASHIONS), CMUL: CONSTANT MULTIPLICATION WHERE NUMS256 IS $16 \cdot 256$-bit $(0x3BEF)$ TED379 IS $24 \cdot 379$-bit $(0x22FCA)$ AND NUMS384 IS $16 \cdot 256$-bit $(0x2D25)$R (UNROLL FASHION)

Operation	ADD	SUB	MUL	SQR	INV	CMUL
p_{256}	47	45	543	428	115,878	67
p_{379}	23	48	1126	884	347,847	85
p_{384}	70	68	1139	898	358,962	101

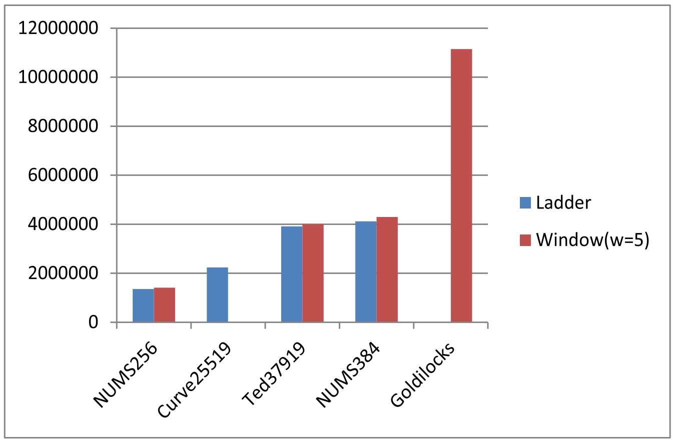

Fig. 3. Comparison of MSR curves with Curve25519 and Goldilock on ARM11 microcontroller.

B. Protection Against Timing and Simple-Side Channel Attacks

The scalar multiplication in variable execution time is vulnerable to timing attack and simple power analysis. To protect against these attacks, the computations should be in constant execution time and regular fashion.[3] All of the multi-precision finite-field arithmetic operations, elliptic curve group

[3]Note: our implementation is SPA resistance in the sense that it does not contain any conditional statements and, therefore, executes always the same sequence of instructions, independent of the value of the operands. We also include a statement that this is not a perfect protection against SPA since differences in the Hamming weight of the processed data may cause small differences in the power consumption profile, which could be exploited in an SPA attack.

TABLE V
EXECUTION TIME OF SCALAR MULTIPLICATION FOR NUMS256, TED37919 AND NUMS384 ON ARM MICROCONTROLLER

Curves	NUMS256	Ted37919	NUMS384	Curve25519	Ed448
Montg. ladder	1,357,795	3,913,321	4,118,369	2,238,000	–
Window($w = 5$)	1,412,947	4,002,787	4,294,175	–	11,148,000

operations, and scalar multiplication of IoT-NUMS curves are implemented in a highly regular fashion (i.e. without `if-then-else` constructs) so that always exactly the same sequence of operations is executed, independent of the actual value of the operands. For the scalar multiplication we take advantage of the generalized scalar recoding introduced with NUMS [14]. Furthermore, the proposed implementations of IoT-NUMS serve a new benchmark for ECC implementation on 8-bit AVR processors and 32-bit ARM11 processors.

VI. CONCLUSIONS

This paper presented high speed implementation of ECC, and we presented, evaluated, and optimized NUMS256, Ted37919, and NUMS384 on 8-bit AVR and 32-bit ARM11 processors. In particular, we introduced an efficient implementation of multi-precision multiplication and squaring for multiplication on 8-bit AVR and ARM micro-controllers. The finite field multiplication and squaring with the length of 256-bit can be accomplished within 6,301/4,489 and 543/428 clock cycles for 8-bit AVR and 32-bit ARM, respectively. The result sets new speed records on an 8-bit AVR and 32-bit ARM processors. And then, we implemented NUMS curves including NUMS256, Ted37919, and NUMS384, which combine the individual computational advantages of the twisted Edward and Montgomery curves. Finally, we achieved record-setting execution times for scalar multiplication over 256, 379, and 384-bit security prime fields. For example, NUMS256 curve only requires 1.357 M cycles on 32-bit ARM11 processor, which is more than 1.6x faster than the widely-used Curve25519.

ACKNOWLEDGMENT

The authors would like to thank Dr. Patrick Longa from Microsoft Research and Dr. Geovandro Pereira from University of Waterloo, Canada for their helpful comments on various aspects of this work.

REFERENCES

[1] E. Wenger, T. Unterluggauer, and M. Werner, “8/16/32 shades of elliptic curve cryptography on embedded processors,” in *Progress in Cryptology—INDOCRYPT*. Cham, Switzerland: Springer, 2013, pp. 244–261.

[2] C. Costello, P. Longa, and M. Naehrig, “A brief discussion on selecting new elliptic curves,” Microsoft Res., Bengaluru, Karnataka, Tech. Rep. MSR-TR-2015-46, 2015.

[3] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, “Comparing elliptic curve cryptography and RSA on 8-bit CPUs,” in *Cryptographic Hardware and Embedded Systems—CHES*. Berlin, Germany: Springer, 2004, pp. 119–132.

[4] H. Seo and H. Kim, “Consecutive operand-caching method for multiprecision multiplication, revisited,” *J. Inf. Commun. Converg. Eng.*, vol. 13, no. 1, pp. 27–35, 2015.

[5] H. Fujii and D. F. Aranha, “Curve25519 for the Cortex-M4 and beyond,” in *Proc. 5th Int. Conf. Cryptol. Inf. Secur. Latin Amer. (LATINCRYPT)*, Habana, Cuba, Sep. 2017, pp. 36–37.

[6] M. Hutter and E. Wenger, “Fast multi-precision multiplication for public-key cryptography on embedded microprocessors,” in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2011, pp. 459–474.

[7] M. Düll *et al.*, “High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers,” *Des., Codes Cryptogr.*, vol. 77, nos. 2–3, pp. 493–514, 2015.

[8] P. Barrett, “Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor,” in *Proc. Conf. Theory Appl. Cryptograph. Techn.*, 1986, pp. 311–323.

[9] M. Hutter and P. Schwabe, “Multiprecision multiplication on AVR revisited,” *J. Cryptograph. Eng.*, vol. 5, no. 3, pp. 201–214, 2014.

[10] H. Seo, Z. Liu, J. Choi, and H. Kim, “Optimized Karatsuba squaring on 8-bit AVR processors,” *Secur. Commun. Netw.*, vol. 8, no. 18, pp. 3546–3554, 2015. [Online]. Available: http://eprint.iacr.org/2014/817.pdf

[11] M. Scott and P. Szczechowiak, “Optimizing multiprecision multiplication for public key cryptography,” IACR Cryptol. ePrint Arch., Tech. Rep. 299, 2007. [Online]. Available: https://eprint.iacr.org/2007/299.pdf

[12] Z. Liu, H. Seo, J. Großschädl, and H. Kim, “Reverse product-scanning multiplication and squaring on 8-bit AVR processors,” in *Proc. Int. Conf. Inf. Commun. Secur.*, 2014, pp. 158–175.

[13] J. W. Bos, C. Costello, P. Longa, and M. Naehrig, “Specification of curve selection and supported curve parameters in MSR ECCLib,” Microsoft Res., Bengaluru, Karnataka, Tech. Rep. MSR-TR-2014-92, 2014.

[14] J. Bos, C. Costello, P. Longa, and M. Naehrig, “Selecting elliptic curves for cryptography: An efficiency and security analysis,” *J. Cryptograph. Eng.*, vol. 6, no. 4, pp. 259–286, 2016.

[15] Z. Liu, P. Longa, G. Pereira, O. Reparaz, and H. Seo, “Four$\mathbb{Q}$ on embedded devices with strong countermeasures against side-channel attacks,” *IEEE Trans. Dependable Secure Comput.*, to be published.

[16] J. Renes, P. Schwabe, B. Smith, and L. Batina, “μKummer: Efficient hyperelliptic signatures and key exchange on microcontrollers,” in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst.*, 2016, pp. 301–320.

[17] D. J. Bernstein, “Curve25519: New diffie-Hellman speed records,” in *Proc. Int. Workshop Public Key Cryptogr.*, 2006, pp. 207–228.

[18] M. Hamburg, “Ed448-Goldilocks, a new elliptic curve,” IACR Cryptol. ePrint Arch., Tech. Rep. 625, 2015. [Online]. Available: https://eprint.iacr.org/2015/625

[19] D. J. Bernstein and T. Lange. (2015). *SafeCurves: Choosing Safe Curves for Elliptic-Curve Cryptography*. [Online]. Available: https://safecurves.cr.yp.to/

[20] J. W. Bos, C. Costello, H. Hisil, and K. Lauter, “Fast cryptography in genus 2,” in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, 2013, pp. 194–210.

[21] C. Costello and P. Longa, “Four$\mathbb{Q}$: Four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime,” in *Advances in Cryptology—ASIACRYPT* (Lecture Notes in Computer Science), vol. 9452, T. Iwata and J. H. Cheon, Eds. Springer, 2015, pp. 214–235. [Online]. Available: https://eprint.iacr.org/2015/565

[22] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, “Twisted Edwards curves,” in *Progress in Cryptology—AFRICACRYPT* (Lecture Notes in Computer Science), vol. 5023, S. Vaudenay, Ed. Springer, 2008, 389–405.

[23] P. Longa, “FourQ NEON: Faster elliptic curve scalar multiplications on ARM processors,” in *Proc. Int. Conf. Sel. Areas Cryptogr.*, 2016, pp. 501–519.

[24] L. Uhsadel, A. Poschmann, and C. Paar, “Enabling full-size public-key algorithms on 8-bit sensor nodes,” in *Proc. Eur. Workshop Secur. Ad-Hoc Sensor Netw.*, 2007, pp. 73–86.

[25] T. Unterluggauer and E. Wenger, “Efficient pairings and ECC for embedded systems,” in *Cryptographic Hardware and Embedded Systems—CHES*. Berlin, Germany: Springer, 2014, pp. 298–315.

Authors’ photographs and biographies not available at the time of publication.