All the Polynomial Multiplication You Need on RISC-V

Hwajeong Seo[1][0000−0003−0069−9061], Hyeokdong Kwon[1], Siwoo Eum[1], Kyungbae Jang[1], Hyunjun Kim[1], Hyunji Kim[1], Minjoo Sim[1], Gyeongju Song[1], and Wai-Kong Lee[2]

[1]IT Department, Hansung University, Seoul (02876), South Korea,
{ hwajeong84, korlethean, shuraatum, starj1023, khj930704, khj1594012, minjoos9797, thdrudwn98}@gmail.com
[2]Department of Computer Engineering,
Gachon University, Seongnam, Incheon (13120), Korea,
waikonglee@gachon.ac.kr

Abstract. Polynomial multiplication is a core operation for public key cryptography, such as pre-quantum cryptography (e.g. elliptic curve cryptography) and post-quantum cryptography (e.g. code-based cryptography and multivariate-based cryptography). For this reason, the efficient and secure implementation of polynomial multiplication has been actively conducted for high availability and security level in application services. In this paper, we present all polynomial multiplication methods on modern 32-bit RISC-V processors. We re-designed expensive implementations of polynomial multiplication on legacy microcontrollers (e.g. 8-bit AVR, 16-bit MSP, and 32-bit ARM) for new instruction sets of 32-bit RISC-V processors. Secondly, we suggest the optimal operand length for each polynomial multiplication on 32-bit RISC-V processors. With this implementation technique and Karatsuba algorithm, we achieved scalable features, which ensures the polynomial multiplication in any operand lengths with reasonably fast performance. Third, we propose instruction set extensions for the optimal implementation of polynomial multiplication on 32-bit RISC-V processors. This new feature introduces significant performance enhancements. Lastly, the proposed implementation is a public domain and following researchers can easily re-produce the result.

Keywords: Secure Polynomial Multiplication · Side Channel Attack · Cache Attack · Constant Timing · RISC-V Processors. · Instruction Set Extensions

1 Introduction

Public key cryptography, such as pre-quantum cryptography (e.g. elliptic curve cryptography) and post-quantum cryptography (e.g. code-based cryptography and multivariate-based cryptography) are based on the polynomial multiplication [1,2]. In particular, the polynomial multiplication has a large computational

load when it performs the operation on the large operand size. Therefore, the efficient and secure polynomial multiplication implementation is very important to improve the performance of pre-quantum cryptography and post-quantum cryptography In this paper, we show all kinds of polynomial multiplication methods on modern 32-bit RISC-V processors. We re-designed expensive implementations of polynomial multiplication on legacy microcontrollers (e.g. 8-bit AVR, 16-bit MSP, and 32-bit ARM) for the state-of-art instruction sets of 32-bit RISC-V processors. Furthermore, we suggested the optimal operand length for each polynomial multiplication on 32-bit RISC-V processors and the scalable design by taking advantages of Karatsuba algorithm. Third, we proposed additional instruction sets for optimized polynomial multiplication implementations on 32-bit RISC-V processors. This new feature can accelerate the performance of polynomial multiplication by reducing clock cycles. Detailed contributions are as follows:

1.1 Contribution

- **First Optimized Implementations of Polynomial Multiplication on 32-bit RISC-V Processors** We optimized two methods for the polynomial multiplication on 32-bit RISC-V processors. In order to achieve the high performance, we exploited RISC-V instructions and carefully utilized general purpose registers of 32-bit RISC-V processors.
- **Optimal and Scalable Implementation** We suggested the optimal operand length for each polynomial multiplication on 32-bit RISC-V processors. With proposed polynomial multiplication and Karatsuba algorithm, we provide scalable features, which ensures any operand lengths with reasonably fast performance.
- **Instruction Set Extension for Polynomial Multiplication** We suggested additional instruction sets for optimal implementation of polynomial multiplication on 32-bit RISC-V processors. We show that with small extension on current instruction sets, the performance can be improved further.
- **Proposed Implementations in Public Domain** We uploaded proposed implementations as an open source. Research followers can utilize the source code to re-produce the result without difficulties.

`https://bit.ly/3Bv8Qf8`

The remainder of the paper is structured as follows: We review the related work on polynomial multiplication, Karatsuba algorithm, and 32-bit RISC-V processors in Section 2. We present the optimized implementation of polynomial multiplication on 32-bit RISC-V processors and instruction extensions for further optimizations in Section 3. In Section 4, we present results on 32-bit RISC-V platforms (i.e. HiFive1 Rev B). We end with conclusions in Section 5.

Table 1. Purpose of registers in 32-bit RISC-V processors.

Register	Description	Register	Description
x0	zero register	tp (x4)	thread pointer
ra (x1)	zero register	a0-a7	function argument & return value
sp (x2)	stack pointer	s0-s11	saved registers
gp (x3)	global pointer	t0-t6	temporal registers

Table 2. Instruction set for 32-bit RISC-V processor used for polynomial multiplication.

Instruction	Description	Instruction	Description
add	add	mv	move
addi	add immediate	sw	store word
sub	subtract	lw	load word
xor	exclusive-or	bne	branch not equal
or	OR	slli	shift left
andi	AND immediate	srli	shift right
mul	multiply	li	load immediate

2 Related Works

2.1 RISC-V Processor

The RISC-V processor is a new computer CPU architecture. This has been developed by UC Berkeley from 2010 [3]. The processor can be utilized for academic, research, and industrial commercialization purposes. The main feature of the RISC-V processor is that the basic instruction set is provided by the consortium, but there are no restrictions on instruction extensions (i.e. customization). By customized instruction sets for certain application services, it is possible to improve the performance, significantly. Recently, many works devoted to improve the performance of cryptography on 32-bit RISC-V processors [4–6]. In this paper we utilized 32-bit RISC-V processors based on RV32IMAC instruction set for high-speed polynomial multiplication operations. Available registers of target RISC-V processors are given in Table 1. In Table 2, the instruction set for 32-bit RISC-V processor used for polynomial multiplication is given.

2.2 Polynomial Multiplication

Block Comb (Conditional Branch) This method executes consecutive bit-wise exclusive-or on intermediate results under the condition of bit setting by block-wise [7]. If the target bit is set to 1, destination registers are updated with operands. Afterward, the intermediate result is left-shifted by 1-bit to align the location of result. Since, whole processes are conducted in a block-wise fashion, the intermediate result and operand bit test are handled, efficiently. There

Algorithm 1 Lopez et al. multiplication in $\mathbb{F}_{2^m}$ [13]

Input: $A = A[0, ..., n-1], B = B[0, ..., n-1]$ where word size is 8-bit.
Output: $C = C[0, ..., 2n-1]$.

1: Compute $T = U \cdot B$ for all polynomials U of degree lower than $t = 4$-bit.
2: $C[0, ..., 2n-1] \leftarrow 0$
3: **for** k from 0 by 1 to $n-1$ **do**
4: $u \leftarrow A[k] \gg t$
5: **for** j from 0 by 1 to $n-1$ **do**
6: $C[j+k] \leftarrow C[j+k] \oplus T(u)[j]$
7: **end for**
8: **end for**
9: $C \leftarrow C \cdot 2^t$
10: **for** k from 0 by 1 to $n-1$ **do**
11: $u \leftarrow A[k] \bmod 2^t$
12: **for** j from 0 by 1 to $n-1$ **do**
13: $C[j+k] \leftarrow C[j+k] \oplus T(u)[j]$
14: **end for**
15: **end for**
16: **return** C

are several variant on the Block-Comb method. In [8], they fully utilized general purpose registers by selecting unbalanced block shape for the computation. In [9], Karatsuba algorithm is applied to the Block-Comb method. This reduces the computation complexity, significantly. In [10], the bitslicing like Block-Comb method is proposed. By re-ordering the operand in bitslicing order, the computation is performed with small number of registers, efficiently. In [11, 12], the polynomial multiplication is implemented on ARMv8 processors by taking advantages of careless multiplication of ARMv8 processors.

Lopez et al.'s Method The look-up table based polynomial multiplication replaces bit-wise operations into simple look-up table accesses [13]. To perform the method, the look-up table by multiplying one operand with certain offsets is calculated and placed into memory (i.e. pre-computed result). The range of offset is usually chosen to 4-bit (`0x0`∼`0xf`) for 16 (2^4) cases. The look-up table occupies memory size at least $16 \times m$, where m is size of operands. Afterward, the look-up table is accessed by 4-bit wise and then updated to intermediate results with the pre-computed result. Detailed descriptions of polynomial multiplication on operands (A and B with n words) are given in Algorithm 1.

Side Channel Resistant Implementation with Conditional Statements The previous LUT based polynomial multiplication (i.e. Lopez et. al.'s method) ensures the implementation with the constant timing. However, the horizontal Correlation Power Analysis (CPA) on weights of LUT can be identified with only a power trace of binary field multiplication. In order to prevent CPA, they

Table 3. Register allocation for look-up table based polynomial multiplication.

Registers	Descriptions
$s0 \sim s7$	intermediate result
$a0 \sim s2$	memory pointer
$a3, t0 \sim t6$	temporal storage
$a4 \sim a7$	operand storage
ra	offset value

suggested a mask based polynomial multiplication [14]. With proposed secure polynomial multiplication, Galois/Counter Mode (GCM) is implemented in a secure way [15]. In [16], they presented concepts of Dummy XOR with garbage registers and instruction level atomicity (ILA), and also present secure binary field (BF) multiplication method using them, which runs in a constant-time and fixed pattern. The method is further improved in [17, 18]

2.3 Asymptotically Faster Multiplication: Karatsuba Multiplication

The basic idea of Karatsuba multiplication is to split a multiplication of two n words operands into three multiplications of size $n/2$, which is possible at the expense of some addition operations [19]. Taking the multiplication of n words operands A and B as an example, we represent operands as $A = A_H \cdot 2^{n/2} + A_L$ and $B = B_H \cdot 2^{n/2} + B_L$. The multiplication $P = A \cdot B$ can be computed according to the following equation

$$P = A \cdot B = A_H \cdot B_H \cdot 2^n + [(A_H + A_L)(B_H + B_L) - A_H \cdot B_H - A_L \cdot B_L] \cdot 2^{n/2} + A_L \cdot B_L$$

3 Proposed Method

3.1 Look-up Table based Polynomial Multiplication

This approach requires the pre-computation on the operand before the polynomial multiplication (See Step 1 of Algorithm 1). For the optimal register utilization, we selected 128-bit wise operands for the polynomial multiplication. This requires 320 bytes (16 cases $\times$ 4 bytes $\times$ 5 words) for the LUT storage. For the efficient look-up table construction, we follow the Algorithm 2. Detailed orders of table construction are as follows:

$$0 \to 1 \to 2 \to 4 \to 3 \to 7 \to 6 \to 5 \to 8 \to 9 \to 11 \to 15 \to 14 \to 12 \to 13 \to 10$$

By following above order, we can cache the intermediate result and re-use them to construct other tables without memory accesses.

Algorithm 2 Table generation for look-up table based polynomial multiplication.

Input: Operands for multiplication.
Output: Pre-computed tables for operands.

1: *Storing zero values to* $0-th$ *table.*
2: *Loading operands to registers* $(s4 \sim s0)$.
3: *Storing operands* $(s4 \sim s0)$ *to first table.*
4: $s9 \sim s5 \leftarrow s4 \sim s0 \ll 1$.
5: *Storing operands* $(s9 \sim s5)$ *to second table.*
6: $t4 \sim t0 \leftarrow s9 \sim s5 \ll 1$.
7: *Storing operands* $(t4 \sim t0)$ *to* $4-th$ *table.*
8: $a7 \sim a3 \leftarrow (s4 \sim s0) \oplus (s9 \sim s5)$.
9: *Storing operands* $(a7 \sim a3)$ *to third table.*
10: $a7 \sim a3 \leftarrow (a7 \sim a3) \oplus (t4 \sim t0)$.
11: *Storing operands* $(a7 \sim a3)$ *to* $7-th$ *table.*
12: $a7 \sim a3 \leftarrow (a7 \sim a3) \oplus (s4 \sim s0)$.
13: *Storing operands* $(a7 \sim a3)$ *to* $6-th$ *table.*
14: $a7 \sim a3 \leftarrow (s4 \sim s0) \oplus (t4 \sim t0)$.
15: *Storing operands* $(a7 \sim a3)$ *to* $5-th$ *table.*
16: $a7 \sim a3 \leftarrow t4 \sim t0 \ll 1$.
17: *Storing operands* $(a7 \sim a3)$ *to* $8-th$ *table.*
18: $a7 \sim a3 \leftarrow (a7 \sim a3) \oplus (s4 \sim s0)$.
19: *Storing operands* $(a7 \sim a3)$ *to* $9-th$ *table.*
20: $a7 \sim a3 \leftarrow (a7 \sim a3) \oplus (s9 \sim s5)$.
21: *Storing operands* $(a7 \sim a3)$ *to* $11-th$ *table.*
22: $a7 \sim a3 \leftarrow (a7 \sim a3) \oplus (t4 \sim t0)$.
23: *Storing operands* $(a7 \sim a3)$ *to* $15-th$ *table.*
24: $a7 \sim a3 \leftarrow (a7 \sim a3) \oplus (s4 \sim s0)$.
25: *Storing operands* $(a7 \sim a3)$ *to* $14-th$ *table.*
26: $a7 \sim a3 \leftarrow (a7 \sim a3) \oplus (s9 \sim s5)$.
27: *Storing operands* $(a7 \sim a3)$ *to* $12-th$ *table.*
28: $a7 \sim a3 \leftarrow (a7 \sim a3) \oplus (s4 \sim s0)$.
29: *Storing operands* $(a7 \sim a3)$ *to* $13-th$ *table.*
30: $a7 \sim a3 \leftarrow t4 \sim t0 \ll 1$.
31: $a7 \sim a3 \leftarrow (a7 \sim a3) \oplus (s9 \sim s5)$.
32: *Storing operands* $(a7 \sim a3)$ *to* $10-th$ *table.*

In Algorithm 3, polynomial multiplication by 4-bit in batch processing (i.e. 4 bytes) is given. In Step $1 \sim 4$, the location of index for look-up table is selected through the right shift operation with immediate values. In Step $5 \sim 8$, lower 4-bit of source registers ($t0 \sim t3$) are extracted with the masking value (15; `0xF`). In Step 9, the index register (`t0`) is multiplied by the offset register (`ra`). In the implementation, the offset register is set to 20 since each index has 5 words for the pre-computation. From Step 11 to 15, pre-computed values are loaded to temporal registers (`s8, s9, s10, s11, t4`). From Step 16 to 20, pre-computed values are added to intermediate results (`s0, s1, s2, s3, s4`). Similar to the first computation, from Step 21 to Step 56, second, third, and fourth computations are performed, subsequently. Lastly, the result is returned.

In Algorithm 4, the shift left operation by 4-bit in batch processing is given. In Step $1 \sim 7$, source registers ($s0 \sim s6$) are shifted to right by 28-bit and stored them into destination registers ($t0 \sim t6$). Afterward, intermediate results are shifted to left by 4-bit in Step $8 \sim 15$. Both results are added together in Step $16 \sim 22$. Lastly, the result is returned.

3.2 Secure Branch based Polynomial Multiplication

Secure branch based polynomial multiplication utilizes the instruction level atomicity. In Algorithm 5, detailed descriptions are given. In Step 1, the operand

Algorithm 3 Polynomial multiplication by 4-bit in batch processing in source code level.

Input: Eight 32-bit registers (`s0`∼`s7`), four registers for operands (`a4`∼`a7`), nine temporal registers (`t0`∼`t4`, `s8`∼`s11`), offset register (`ra`).
Output: Eight 32-bit registers (`s0`∼`s7`).

```
//Index extraction
 1: srli t0, a4, offset
 2: srli t1, a5, offset
 3: srli t2, a6, offset
 4: srli t3, a7, offset

 5: andi t0, t0, 15
 6: andi t1, t1, 15
 7: andi t2, t2, 15
 8: andi t3, t3, 15

//First computation
 9: mul t0, t0, ra
10: add t0, a3, t0

11: lw s8, 4*0(t0)
12: lw s9, 4*1(t0)
13: lw s10, 4*2(t0)
14: lw s11, 4*3(t0)
15: lw t4, 4*4(t0)

16: xor s0, s0, s8
17: xor s1, s1, s9
18: xor s2, s2, s10
19: xor s3, s3, s11
20: xor s4, s4, t4

//Second computation
21: mul t1, t1, ra
22: add t1, a3, t1

23: lw s8, 4*0(t1)
24: lw s9, 4*1(t1)
25: lw s10, 4*2(t1)
26: lw s11, 4*3(t1)
27: lw t4, 4*4(t1)

28: xor s1, s1, s8
29: xor s2, s2, s9
30: xor s3, s3, s10
31: xor s4, s4, s11
32: xor s5, s5, t4

//Third computation
33: mul t2, t2, ra
34: add t2, a3, t2

35: lw s8, 4*0(t2)
36: lw s9, 4*1(t2)
37: lw s10, 4*2(t2)
38: lw s11, 4*3(t2)
39: lw t4, 4*4(t2)

40: xor s2, s2, s8
41: xor s3, s3, s9
42: xor s4, s4, s10
43: xor s5, s5, s11
44: xor s6, s6, t4

//Fourth computation
45: mul t3, t3, ra
46: add t3, a3, t3

47: lw s8, 4*0(t3)
48: lw s9, 4*1(t3)
49: lw s10, 4*2(t3)
50: lw s11, 4*3(t3)
51: lw t4, 4*4(t3)

52: xor s3, s3, s8
53: xor s4, s4, s9
54: xor s5, s5, s10
55: xor s6, s6, s11
56: xor s7, s7, t4

57: return s0∼s7
```

setting is tested with the logical-AND instruction on operand ($a0$) and masking value (ra). When the bit is reset (i.e. 0), the program counter is added with offset and moves to Step 9. Afterward, operands ($a4 \sim a7$) are added to the intermediate results ($s0 \sim s3$). In Step 14, the program counter is moved to label 2. If the bit is set, it moves to Step 3 and performs dummy addition operations, which emulate addition operations on the program counter. In Step 4 $\sim$ 7, it emulates dummy exclusive-or operations (i.e. Step 10 $\sim$ 13). Afterward, the program counter is moved to label 2. As we explained above, both routines perform same operations. There is no memory operation depending on certain information. This ensures high side channel resistance features. From Step 16 to 60, similar operations on other operands are performed, subsequently. From Step 61 to 82, intermediate results are shifted to left by 1-bit. In Step 83, the offset register is also shifted to right by 1-bit. Finally, the intermediate result is returned. The register utilization is given in Table 4.

Algorithm 4 Shift left by 4-bit in batch processing in source code level.

Input: Eight 32-bit registers (`s0`∼`s7`), seven temporal registers (`t0`∼`t6`).
Output: Eight 32-bit registers (`s0`∼`s7`).

```
//Right shift
 1: srli t0, s0, 28
 2: srli t1, s1, 28
 3: srli t2, s2, 28
 4: srli t3, s3, 28
 5: srli t4, s4, 28
 6: srli t5, s5, 28
 7: srli t6, s6, 28
//Left shift
 8: slli s0, s0, 4
 9: slli s1, s1, 4
10: slli s2, s2, 4
11: slli s3, s3, 4
12: slli s4, s4, 4
13: slli s5, s5, 4
14: slli s6, s6, 4
15: slli s7, s7, 4
//Combining both values
16: or s1, s1, t0
17: or s2, s2, t1
18: or s3, s3, t2
19: or s4, s4, t3
20: or s5, s5, t4
21: or s6, s6, t5
22: or s7, s7, t6
23: return s0∼s7
```

Table 4. Register allocation for secure branch based polynomial multiplication.

Registers	Descriptions
$s0 \sim s7$	intermediate result
$a0 \sim a7$	operand storage
$s8 \sim s11$, $t0 \sim t6$	temporal storage
ra	offset value

3.3 Instruction Set Extensions for Polynomial Multiplication

The RISC-V processor supports user defined instruction set to accelerate the target applications. In this Section, we suggest several instruction sets in proof of concept, which are useful for the polynomial multiplication. Proposed instruction set extensions are as follows:

- `srliandi` (`srli + andi`; Step 1 $\sim$ 8 of Algorithm 3): extracting the 4-bit out of certain location of source register.
- `muladd` (`mul + add`; Step 9 $\sim$ 10 of Algorithm 3): calculating the memory address with base pointer and offset value.
- `sllior` (`slli + or`; Step 8 $\sim$ 22 of Algorithm 4): logical-oring with left shifted value.
- `andbnexadd` (`and + bnez + add`; Step 1 $\sim$ 2 of Algorithm 5): masked branching with instruction level atomicity.

In Table 5, performance improvements with instruction set extension for both polynomial multiplication methods are given. For the look-up table based polynomial multiplication, the number of instruction is reduced by 12.7% (826 $\rightarrow$ 721). This is achieved through new extensions such as `srliandi`, `muladd`, and `sllior`. For the secure branch based polynomial multiplication, the number of instruction is reduced by 19.4% ($2,437 \rightarrow 1,964$). This is achieved through new extensions such as `sllior` and `andbnexadd`.

Algorithm 5 Secure polynomial multiplication by 4-bit in batch processing in source code level.

Input: Eight 32-bit registers (`s0`∼`s7`), four registers for operands (`a4`∼`a7`), seven temporal registers (`t0`∼`t6`), offset register (`ra`).
Output: Eight 32-bit registers (`s0`∼`s7`).

```
//First computation
 1: and t0, a0, ra
 2: bnez t0, 1f
 3: add s8, s8, s9
 4: xor s8, s8, a4
 5: xor s9, s9, a5
 6: xor s10, s10, a6
 7: xor s11, s11, a7
 8: j 2f
 9: 1:
10: xor s0, s0, a4
11: xor s1, s1, a5
12: xor s2, s2, a6
13: xor s3, s3, a7
14: j 2f
15: 2:

//Second computation
16: and t0, a1, ra
17: bnez t0, 1f
18: add s8, s8, s9
19: xor s8, s8, a4
20: xor s9, s9, a5
21: xor s10, s10, a6
22: xor s11, s11, a7
23: j 2f
24: 1:
25: xor s1, s1, a4
26: xor s2, s2, a5
27: xor s3, s3, a6
28: xor s4, s4, a7
29: j 2f
30: 2:

//Third computation
31: and t0, a2, ra
32: bnez t0, 1f
33: add s8, s8, s9
34: xor s8, s8, a4
35: xor s9, s9, a5
36: xor s10, s10, a6
37: xor s11, s11, a7
38: j 2f
39: 1:
40: xor s2, s2, a4
41: xor s3, s3, a5
42: xor s4, s4, a6
43: xor s5, s5, a7
44: j 2f
45: 2:

//Fourth computation
46: and t0, a3, ra
47: bnez t0, 1f
48: add s8, s8, s9
49: xor s8, s8, a4
50: xor s9, s9, a5
51: xor s10, s10, a6
52: xor s11, s11, a7
53: j 2f
54: 1:
55: xor s3, s3, a4
56: xor s4, s4, a5
57: xor s5, s5, a6
58: xor s6, s6, a7
59: j 2f
60: 2:

//Shift to left by 1-bit
61: srli t0, s0, 31
62: srli t1, s1, 31
63: srli t2, s2, 31
64: srli t3, s3, 31
65: srli t4, s4, 31
66: srli t5, s5, 31
67: srli t6, s6, 31
68: slli s0, s0, 1
69: slli s1, s1, 1
70: slli s2, s2, 1
71: slli s3, s3, 1
72: slli s4, s4, 1
73: slli s5, s5, 1
74: slli s6, s6, 1
75: slli s7, s7, 1
76: or s1, s1, t0
77: or s2, s2, t1
78: or s3, s3, t2
79: or s4, s4, t3
80: or s5, s5, t4
81: or s6, s6, t5
82: or s7, s7, t6
83: srli ra, ra, 1
84: return s0∼s7
```

4 Evaluation

We utilized a HiFive1 development board as a benchmarking platform. The board contains the FE310-G000 SoC with an E31 core and support the RV32IMAC instruction set. The E31 core is designed as a 5-stage single-issue in-order pipelined CPU@320 MHz. The core has 16 KiB of DTIM memory that is used as RAM. To

Table 5. Performance improvements with instruction set extensions for both polynomial multiplication methods. Notations (LUT, Secure, and w) represent look-table based polynomial multiplication, secure branch based polynomial multiplication, and with RISC-V extension, respectively.

Instruction	LUT	LUT^w	Secure	Secure^w
`slli`	69	20	249	32
`srli`	86	62	248	248
`or`	65	16	217	0
`xor`	215	215	1,024	1,024
`mul`	32	0	0	0
`add`	32	0	128	0
`addi`	4	4	4	4
`lw`	181	181	22	22
`sw`	101	101	22	22
`and`	0	0	128	0
`andi`	28	4	0	0
`li`	12	12	9	9
`mv`	1	1	2	2
`bnez`	0	0	128	0
`j`	0	0	256	256
`srliandi`	0	24	0	0
`muladd`	0	32	0	0
`sllior`	0	49	0	217
`andbnexadd`	0	0	0	128
Total	826	721	2,437	1,964

accelerate instruction fetches from the flash memory, the E31 comes with 16 KiB of 2-way instruction cache. The evaluation process is performed on SiFive Eclipse IDE for C/C++ Development (version 4.7.2.) with gcc-8.3.0 and optimization level 2 (`-O2`).

In Table 6, the evaluation of polynomial multiplication on target 32-bit RISC-V processors is given. Various operand lengths ranging from 128-bit to 512-bit are evaluated. For implementations of 256-bit and 512-bit polynomial multiplication operations, 1-level and 2-level Karatsuba algorithms are utilized, respectively. Together with the efficiency, secure countermeasures against side channel attack are also considered. Between look-up table based approach and secure branch based approach, look-up table based approach shows higher performance by replacing the expensive bit-wise operation into memory accesses than secure branch based approach by 51%, 53%, and 54% for 128-bit, 256-bit, and 512-bit, respectively. The implementation is constant timing but it is is known to be vulnerable to cache-based timing attacks [20, 21] and CPA. A CPU cache can leak information about which memory address has been accessed during a

Table 6. Evaluation of polynomial multiplication on 32-bit RISC-V processors depending on operand size in terms of timing (clock cycles). Notations ($c1$, $c2$, and $c3$) indicate constant timing, cache-free, and correlation power analysis resistance, respectively.

Method	Language	128-bit	256-bit	512-bit	Side Channel Resistance
Look-up Table	assembly	1,026	3,425	10,950	$c1$
Secure Branch	assembly	1,989	6,412	20,056	$c1$, $c2$, $c3$

computation. When this memory address depends on a secret intermediate value as is the case with the table approach, it can be used to extract secret information Our benchmarking platform does not have a data cache. Therefore, it should be safe to use a table-based polynomial multiplication implementation on this device. For the CPA attack, it requires highly controlled experimental setting to extract the information. For this reason, the look-up table based approach is reasonably secure implementation in real world applications.

Alternative implementation technique (namely secure branch based implementation) shows slower performance than look-up table based approach. However, it achieved instruction level atomicity. For this reason, it ensures highest security levels with constant timing, cache-free, and correlation power analysis resistance among polynomial multiplication methods.

5 Conclusion

In this paper, we presented the polynomial multiplication on 32-bit RISC-V processors. Main contributions of our works can be summarized in the following four aspects.

First, we firstly implemented the polynomial multiplication on legacy microcontrollers (e.g. 8-bit AVR, 16-bit MSP, and 32-bit ARM) for new instruction sets of 32-bit RISC-V processors. Second, we suggested the optimal operand length for each polynomial multiplication on 32-bit RISC-V processors. With this polynomial multiplication implementation and Karatsuba algorithm, we achieved the scalable implementation, which ensures any operand lengths for the polynomial multiplication operation with reasonably fast performance. Third, we suggested additional instruction sets for the optimal implementation of the polynomial multiplication operation on 32-bit RISC-V processors. This new feature can improve the performance further. Lastly, the proposed implementation is public domain and following researchers can easily re-produce the result in this paper.

As a future work, we will further explore the optimization of binary field multiplication and efficient instruction set extensions for fast computations of modern cryptography.

References

1. H. Seo, J. Kim, J. Choi, T. Park, Z. Liu, and H. Kim, "Small private key MQPKS on an embedded microprocessor," *Sensors*, vol. 14, no. 3, pp. 5441–5458, 2014.

2. K.-A. Shim, C.-M. Park, N. Koo, and H. Seo, "A high-speed public-key signature scheme for 8-b IoT-constrained devices," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3663–3677, 2020.
3. K. Asanovic and A. Waterman, "The RISC-V instruction set manual," in *Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*, vol. 2, RISC-V Foundation, 2019.
4. H. Kwon, H. Kim, S. Eum, M. Shim, H. Kim, W.-K. Lee, Z. Hu, and H. Seo, "Optimized implementation of SM4 on AVR microcontrollers, RISC-V processors, and ARM processors,"
5. H. Seo, H. Kwon, K. Jang, and H. Kim, "Optimized implementation of scalable multi-precision multiplication method on RISC-V processor for high-speed computation of post-quantum cryptography," *Journal of the Korea Institute of Information Security & Cryptology*, vol. 31, no. 3, pp. 473–480, 2021.
6. K. Stoffelen, "Efficient cryptography on the RISC-V architecture," in *International Conference on Cryptology and Information Security in Latin America*, pp. 323–340, Springer, 2019.
7. M. Shirase, Y. Miyazaki, T. Takagi, D.-G. Han, and D. Choi, "Efficient implementation of pairing-based cryptography on a sensor node," *IEICE transactions on information and systems*, vol. 92, no. 5, pp. 909–917, 2009.
8. H. Seo, Y. Lee, H. Kim, T. Park, and H. Kim, "Binary and prime field multiplication for public key cryptography on embedded microprocessors," *Security and Communication Networks*, vol. 7, no. 4, pp. 774–787, 2014.
9. H. Seo, Z. Liu, J. Choi, and H. Kim, "Karatsuba–Block-Comb technique for elliptic curve cryptography over binary fields," *Security and Communication Networks*, vol. 8, no. 17, pp. 3121–3130, 2015.
10. S. C. Seo and H. Seo, "Highly efficient implementation of NIST-compliant Koblitz curve for 8-bit AVR-based sensor nodes," *IEEE Access*, vol. 6, pp. 67637–67652, 2018.
11. H. Seo, Z. Liu, Y. Nogami, J. Choi, and H. Kim, "Binary field multiplication on ARMv8," *Security and Communication Networks*, vol. 9, no. 13, pp. 2051–2058, 2016.
12. H. Seo, "Faster (feat. ECC pmull) over F_2^{571}," in *A Systems Approach to Cyber Security: Proceedings of the 2nd Singapore Cyber-Security R&D Conference (SG-CRC 2017)*, vol. 15, p. 97, IOS Press, 2017.
13. J. López and R. Dahab, "High-speed software multiplication in $f(2^m)$," in *International Conference on Cryptology in India*, pp. 203–212, Springer, 2000.
14. H. Seo, C.-N. Chen, Z. Liu, Y. Nogami, T. Park, J. Choi, and H. Kim, "Secure binary field multiplication," in *International Workshop on Information Security Applications*, pp. 161–173, Springer, 2015.
15. Z. Liu, H. Seo, C.-N. Chen, Y. Nogami, T. Park, J. Choi, and H. Kim, "Secure GCM implementation on AVR," *Discrete Applied Mathematics*, vol. 241, pp. 58–66, 2018.
16. S. C. Seo and H. Kim, "SCA-resistant GCM implementation on 8-bit AVR microcontrollers," *IEEE Access*, vol. 7, pp. 103961–103978, 2019.
17. S. C. Seo and D. Kwon, "Highly efficient SCA-resistant binary field multiplication on 8-bit AVR microcontrollers," *Applied Sciences*, vol. 10, no. 8, p. 2821, 2020.
18. K. Kim, S. Choi, H. Kwon, H. Kim, Z. Liu, and H. Seo, "PAGE—practical AES-GCM encryption for low-end microcontrollers," *Applied Sciences*, vol. 10, no. 9, p. 3131, 2020.

19. A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, pp. 293–294, Russian Academy of Sciences, 1962.
20. D. J. Bernstein, "Cache-timing attacks on AES," 2005.
21. D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' track at the RSA conference*, pp. 1–20, Springer, 2006.