241
Journal of The Korea Institute of Information Security & Cryptology
VOL.25, NO.2, Apr. 2015
ISSN 1598-3986(Print)
ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2015.25.2.241

8-bit ATmega128 프로세서 환경에 최적화된 이진체 감산 알고리즘*

박 동 원,† 권 희 택, 홍 석 희‡
고려대학교 정보보호연구원

Optimized Binary Field Reduction Algorithm on 8-bit ATmega128 Processor*

Dong-won Park,† Heetaek Kwon, Seokhie Hong‡
Center for Information Security Technologies(CIST), Korea University

요 약

유한체 연산을 기반으로 하는 공개키 암호 시스템은 고속 연산이 매우 중요한 과제이다. 본 논문에서는 8-bit ATmega128 프로세서 환경에서 이진 기약다항식 $f(x)= x^{271}+x^{207}+x^{175}+x^{111}+1$과 $f(x)= x^{193}+x^{145}+x^{129}+x^{113}+1$을 이용한 감산 연산의 효율성을 높이는 데에 중점을 두었다. 기존의 감산 연산 알고리즘인 Fast reduction의 최종적인 감산 결과 값을 제시함으로써, 중복 발생하는 메모리 접근을 최소화 하여 최적화된 감산 알고리즘을 제시한다. 제안하는 기법을 어셈블리 언어로 구현 시 기존의 감산 연산 알고리즘과 비교하여 각각 53%, 55% 향상된 결과를 얻었다.

ABSTRACT

In public-key cryptographic system based on finite field arithmetic, it is very important to challenge for implementing high speed operation. In this paper, we focused on 8-bit ATmega128 processor and concentrated on enhancing efficiency of reduction operation which uses irreducible polynomial $f(x)= x^{271}+x^{207}+x^{175}+x^{111}+1$ and $f(x)= x^{193}+x^{145}+x^{129}+x^{113}+1$. We propose optimized reduction algorithms which are designed to reduce repeated memory accesses by calculating final reduced values of Fast reduction. There are 53%, 55% improvement when proposed algorithm is implemented using assembly language, compare to previous Fast reduction algorithm.

Keywords: ATmega128 processor, Fast reduction, Efficient implementation

I. 서 론

공개키 기반의 암호 시스템은 대칭키 기반의 암호 시스템에 비하여 상대적으로 속도가 느리다는 단점을 가지고 있다. 하지만 대칭키 암호 시스템은 통신하는 모든 사람에 대해서 새로운 키가 필요하므로 키관리가 어렵다는 특징이 있다. 하지만 공개된 키를 이용하면 인증, 서명 등의 서비스를 대칭키 암호 시스템보다 적은 수의 키로 이용할 수 있다는 장점을 가지고 있다. 공개키 암호 시스템에는 RSA, Elliptic

접수일(2014년 10월 6일), 수정일(1차: 2015년 1월 5일, 2차: 2015년 2월 10일), 게재확정일(2015년 2월 16일)
* 이 논문은 2014년도 정부(미래창조과학부)의 재원으로 한국연구재단-차세대정보 · 컴퓨팅기술개발사업의 지원을 받아 수행된 연구입니다(No. NRF-2014M3C4A7030648).
† 주저자, wony86a@nate.com
‡ 교신저자, shhong@korea.ac.kr(Corresponding author)

Curve Cryptography(ECC)등이 있다. 일반적으로 공개키 암호 시스템은 유한체 연산에 의하여 구성된다. 따라서 유한체 연산의 고속화는 공개키 암호 시스템에서 매우 중요한 과제이고 활발히 연구 되고 있다[1,2]. 그 중에서도 감산 연산은 효율적인 기약다항식을 선택하여 빠르게 구현이 가능하다. 이진체 상에서 곱셈 연산과 제곱 연산의 결과는 입력 값의 두 배 워드 크기를 가지며 이 값을 기준으로 감산 연산이 이루어진다. 본 논문에서 사용한 $GF(2^{271})$, $GF(2^{193})$에서의 감산 연산은 삼항 혹은 오항 기약다항식을 이용하여 시프트 연산과 XOR(Exclusive-OR) 연산으로 구현 가능하다. 본 논문에서는 8-bit 프로세서 환경에서의 감산 연산 최적화를 위해 $GF(2^{271})$상의 기약 다항식 $f(x)=x^{271}+x^{207}+x^{175}+x^{111}+1$과 $GF(2^{193})$상에서의 기약다항식 $f(x)= x^{193}+x^{145}+x^{129}+x^{113}+1$을 선택하였다.

기존의 Fast reduction 알고리즘은 매 반복문에서 한 번에 한 워드씩 감산을 진행한다[4]. 그리고 기약다항식의 형태에 따라 시프트 연산과 XOR 연산의 사용횟수와 위치가 달라지고, 매번 XOR 연산이 적용될 위치의 워드 메모리를 호출하고 저장한다.

본 논문에서는 기존 감산 알고리즘의 반복문 사용방식과는 달리 사전 연산을 통해 중복되어 사라지는 연산들을 제외하여 최종적인 감산 연산 결과 값을 해당 기약 다항식에 대해서 감산 알고리즘 테이블로 각각 Table 2, Table 4로 나타내었다. 각 Table에는 중복되는 워드들의 조합이 많이 발생한다는 사실을 발견할 수 있다. 실제 구현에서 중복 조합들을 한번 계산으로 2번 이상 재사용하는 방식으로 스케줄링하여 최종적인 연산량과 레지스터 사용을 효율적으로 줄일 수 있었다.

제한된 환경에서의 감산 연산을 구현하기 위해 어셈블러 언어로 설계하여 26개의 레지스터를 적극적으로 활용하였고, 기본적으로 제공하는 ATmega128의 rotate shift 연산을 효율적으로 사용하여 감산 연산의 최적화를 이루었다. 결과적으로 최종적인 감산 결과 값에 나타난 중첩되는 워드 조합과 이를 이용한 레지스터의 효율적인 스케줄링, rotate 연산의 적극적인 활용 등을 통해 메모리 접근 연산의 최소화를 가능하게 하였다.

II. 사전지식

2.1 기약다항식

이진체 $GF(2^m)$의 원소들은 다항식 기저 표현법에 의해 bit들의 나열로 표현할 수 있다. 각 bit들은 최고차항이 $(m-1)$ 이하인 다항식의 계수들을 나타낸다. 이진체의 원소들은 기약다항식에 따라 구성방식이 달라지며 각 원소들끼리의 곱셈 연산은 기약다항식을 이용한 감산 연산을 통해서 수행 될 수 있다.

기약다항식으로 삼항다항식 x^m+x^a+1이나 오항다항식 $x^m+x^a+x^b+x^c+1$을 주로 선택한다. 기약다항식을 선택하는 과정은 다음과 같다. 만약 삼항 기약다항식이 존재한다면 그 중에서 가장 차수가 낮은 a를 선택하고 삼항 기약다항식이 존재하지 않으면 오항 기약다항식에서 차수가 가장 낮은 a를 선택한다. 이어서 b,c도 차수가 가장 낮은 것으로 기약다항식을 선택하게 된다. 위와 같은 방식으로 모든 m에 대해 기약다항식을 선택할 수 있다[4].

2.2 ATmega128 프로세서

ATmega128 프로세서는 무선 센서네트워크 환경에서 대표적으로 사용되는 MICA2/MICAz 센서 모트에 탑재된 8-bit 기반의 마이크로프로세서이다. ATmega128 프로세서는 성능과 병렬성을 최대화하기 위하여 프로그램과 데이터를 위한 메모리와 버스를 구분하는 Havard 구조이며 프로그램 메모리 내의 명령어들은 하나의 명령어가 수행될 때, 다음에 수행될 명령어가 프로그램 메모리로부터 prefetch되는 한 단계의 파이프라인으로 처리된다. 이 방법을 통하여 ATmega128 프로세서는 매 클럭 사이클마다 명령어를 수행할 수 있다[6].

ALU(Arithmetic and Logic Unit)가 1클럭 사이클 안에 범용 레지스터의 데이터 대하여 명령어가 수행되고 그 결과 값이 다시 레지스터에 저장된다. 32개의 레지스터 중 6개의 레지스터(R31:R30, R29:R28, R27:R26)는 세 개의 16-bit 간접주소 레지스터(X, Y, Z)로 사용되어 효율적인 주소 계산이 가능하다. 나머지 26개의 레지스터들을 이용하여 유한체 연산을 수행할 수 있다.

	7	0	Addr.	
	R0		\$00	
	R1		\$01	
	R2		\$02	
	...			
	R13		\$0D	
General	R14		\$0E	
Purpose	R15		\$0F	
Working	R16		\$10	
Registers	R17		\$11	
	...			
	R26		\$1A	X-register Low Byte
	R27		\$1B	X-register High Byte
	R28		\$1C	Y-register Low Byte
	R29		\$1D	Y-register High Byte
	R30		\$1E	Z-register Low Byte
	R31		\$1F	Z-register High Byte

Fig. 1. ATmega128 general purpose resisters

III. 기존의 감산 알고리즘

이진체에서 가장 효율적으로 알려진 감산 알고리즘은 기약다항식에 의존하여 알고리즘을 구성하는 Fast reduction 알고리즘이다.

Fast reduction 알고리즘은 루프 한 번에 한 워드씩 감산을 진행한다. 예를 들어 32-bit 환경에서 기약 다항식이 $f(x)=x^{163}+x^7+x^6+x^3+1$ 일 때 입력 값의 10번째 워드(C[9] $=x^{319}+\cdots+x^{288}$)를 다음과 같이 표현할 수 있다.

$$
\begin{aligned}
x^{288} &\equiv x^{132}+x^{131}+x^{128}+x^{125} \bmod(f(x)) \\
x^{289} &\equiv x^{133}+x^{132}+x^{129}+x^{126} \bmod(f(x)) \\
&\vdots \\
x^{319} &\equiv x^{163}+x^{162}+x^{159}+x^{156} \bmod(f(x))
\end{aligned}
$$

위 합동식의 오른쪽 4열을 고려해볼 때, 'C[9]'의

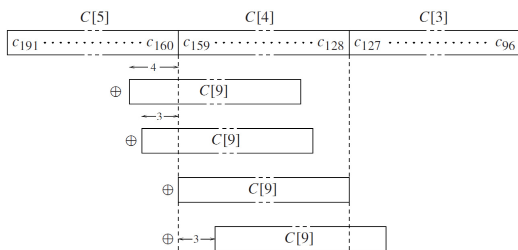

Fig. 2. Reducing the 32-bit word C[9] modulo $f(x)=x^{163}+x^7+x^6+x^3+1$

Algorithm 1. Fast reduction modulo $f(x)=x^{163}+x^7+x^6+x^3+1$ (with 8-bit processor)

Input: $c(x)$ which of degree is at most 324

Output: $c(x)$ $(\bmod\ x^{163}+x^7+x^6+x^3+1)$

1: for i from 41 downto 21 do
  1.1: T ← C[i].
  1.2: C[i-21] ← C[i-21] ⊕ (T<<5).
  1.3: C[i-20] ← C[i-20] ⊕ (T<<4) ⊕ (T<<3) ⊕ T ⊕ (T>>3).
  1.4: C[i-19] ← C[i-19] ⊕ (T>>4) ⊕ (T>>5).
2: T ← C[20]>>3.
3: C[0] ← C[0] ⊕ (T<<7) ⊕ (T<<6) ⊕ (T<<3) ⊕ T.
4: C[1] ← C[1] ⊕ (T>>1) ⊕ (T>>2).
5: C[20] ← C[20]&0x7.
6. Return (C[20],⋯,C[1],C[0])

감산 연산은 해당하는 하위 워드 'C[5],C[4],C[3]'에 7번의 XOR 연산으로 처리가 가능하다. Fig 2에서 보이는 것과 같이 'C[9]'의 최하위 bit는 각각 132, 131, 128, 125번 째 bit들에 XOR 연산으로 처리된다.

Algorithm 1은 8-bit 프로세서 환경에서 곱셈 연산이나 제곱 연산의 결과로 워드 사이즈가 2배로 늘어난 값을 입력으로 하여 기약다항식 $f(x)=x^{163}+x^7+x^6+x^3+1$ 에 대한 Fast reduction 알고리즘을 나타낸 것이다. 알고리즘에 표기된 ⊕ 기호는 bit별 XOR 연산, & 기호는 bit별 AND 연산을 의미하며 <<는 bit left shift 연산을 >>는 bit right shift 연산을 나타낸다.

Fast reduction에서는 매 루프마다 4개의 워드를 LOAD하고 STORE연산이 일어난다. 다음 루프에서는 이전에 사용한 4개의 워드 중 2개를 다시 LOAD하고 STORE한다. 이와 같은 중복연산을 줄이기 위해 [3]에서는 4번의 루프를 한 번에 처리하였다. 따라서 한번 사용한 워드를 다음 루프에서 다시 사용하는 빈도수를 Fast reduction보다 줄였다. Algorithm 2는 [3]에서 제시한 감산연산을 나타낸다.

Algorithm 2. Seo's reduction modulo $f(x) = x^{163}+x^7+x^6+x^3+1$ (with 8-bit processor)

Input: $c(x)$ which of degree is at most 324
Output: $c(x) \pmod{x^{163}+x^7+x^6+x^3+1}$

1: for i ←41 to 21 decrements i by 4 do
  1.1: T_1 ← C[i], T_2 ← C[i-1], T_3 ← C[i-2], T_4 ← C[i-3].
  1.2: C[i-24] ← C[i-24] ⊕ ($T_4 \ll 5$).
  1.3: C[i-23] ← C[i-23] ⊕ ($T_3 \ll 5$) ⊕ ($T_4 \ll 4$) ⊕ ($T_4 \ll 3$) ⊕ T_4 ⊕ ($T_4 \gg 3$).
  1.4: C[i-22] ← C[i-22] ⊕ ($T_3 \ll 5$) ⊕ ($T_3 \ll 4$) ⊕ ($T_3 \ll 3$) ⊕ T_3 ⊕ ($T_3 \gg 3$) ⊕ ($T_4 \gg 4$) ⊕ ($T_4 \gg 5$).
  1.5: C[i-21] ← C[i-21] ⊕ ($T_1 \ll 5$) ⊕ ($T_2 \ll 4$) ⊕ ($T_2 \ll 3$) ⊕ T_2 ⊕ ($T_2 \gg 3$) ⊕ ($T_3 \gg 4$) ⊕ ($T_3 \gg 5$).
  1.6: C[i-20] ← C[i-20] ⊕ ($T_1 \ll 4$) ⊕ ($T_1 \ll 3$) ⊕ T_1 ⊕ ($T_1 \gg 3$) ⊕ ($T_2 \gg 4$) ⊕ ($T_2 \gg 5$).
  1.7: C[i-19] ← C[i-19] ⊕ ($T_1 \gg 4$) ⊕ ($T_1 \gg 5$).
2: T_1 ← C[21], T_2 ← C[20].
3: C[0] ← C[0] ⊕ ($T_1 \ll 5$) ⊕ ($T_2 \ll 4$) ⊕ ($T_2 \ll 3$) ⊕ T_2 ⊕ ($T_2 \gg 3$).
4: C[1] ← C[1] ⊕ ($T_1 \ll 4$) ⊕ ($T_1 \ll 3$) ⊕ T_1 ⊕ ($T_1 \gg 3$) ⊕ ($T_2 \gg 4$) ⊕ ($T_2 \gg 5$).
5: C[2] ← C[2] ⊕ ($T_1 \gg 5$) ⊕ ($T_1 \gg 4$).
6: C[20] ← C[20]&0x7.
7. Return (C[20],⋯,C[1],C[0])

IV. 제안하는 감산 알고리즘

4.1 GF(2^{271}) 상에서의 감산 알고리즘

8-bit ATmega128 프로세서 환경에서 Fast reduction 알고리즘을 기약다항식 $f(x) = x^{271}+x^{207}+x^{175}+x^{111}+1$에 적용하면 Algorithm 3과 같다. 8-bit 환경에서 271-bit 원소는 34개의 8-bit 워드로 구성된다. 유한체 감산 연산 시에는 유한체 곱셈 연산과 유한체 제곱 연산의 결과를 입력받기 때문에 최대 68개의 워드를 입력받게 된다. 따라서 'C[67]'부터 'C[0]'까지 입력으로 받아 감산연산 결과로 'C[33]'부터 'C[0]'까지 출력한다.

Table 1은 Algorithm 3의 루프 카운터 i가 66에서 62까지 동작할 때 수행되는 연산을 기술한 것이다. Table 1에서 밑줄로 표기 된 연산은 데이터

Algorithm 3. Fast reduction modulo $f(x) = x^{271}+x^{207}+x^{175}+x^{111}+1$ (with 8-bit processor)

Input: $c(x)$ which of degree is at most 540
Output: $c(x) \pmod{x^{271}+x^{207}+x^{175}+x^{111}+1}$

1: T ← C[67].
2: C[59] ← C[59] ⊕ T.
3: C[55] ← C[55] ⊕ T.
4: C[47] ← C[47] ⊕ T.
5: C[33] ← C[33] ⊕ (T $\ll$ 1).
6: for i from 66 downto 34 do
  6.1: T ← C[i].
  6.2: C[i-8] ← C[i-8] ⊕ T.
  6.3: C[i-12] ← C[i-12] ⊕ T.
  6.4: C[i-20] ← C[i-20] ⊕ T.
  6.5: C[i-33] ← C[i-33] ⊕ (T $\gg$ 7).
  6.6: C[i-34] ← C[i-34] ⊕ (T $\ll$ 1).
7: T ← C[33]&0x80.
8: C[25] ← C[25] ⊕ T.
9: C[21] ← C[21] ⊕ T.
10: C[13] ← C[13] ⊕ T.
11: C[0] ← C[0] ⊕ (T $\gg$ 7).
12: C[33] ← C[33]&0x7F.
13. Return (C[33],⋯,C[0])

가 메모리로부터 중복 로드하거나 결과 값을 중복 저장하는 부분을 표시한 것이다. Table 1에서 밑줄(__)로 표기 된 'C[32]','C[31]','C[30]','C[29]'은 5번의 루프를 수행할 때 각각 두 번씩 메모리로부터 로드하고 다시 저장한다. 이와 같이 중복 로드하거나 저장하는 값들을 레지스터에서 재사용하여 메모리 접근으로 인한 부하를 줄일 수 있다.

또한 루프 카운터 i가 66일 때와 i가 62일 때 밑줄(__)로 표기 된 'C[54]'를 각각 로드하고 다시 저장한다. 이처럼 루프 카운터가 진행됨에 따라서 동일한 워드를 다시 호출하는 경우가 발생하게 된다. 따라서 이를 효율적으로 처리할 수 있는 방법이 필요하다.

감산 대상의 상위 워드를 살펴보면 x^{272}부터 차례로 8-bit씩 각각 하나의 워드를 형성하게 된다. 여기서 각 워드들은 오직 1-bit left shift와 워드 시프트 연산을 통해서 감산이 가능하다는 특징이 있다. 예를 들어, $x^{272} = x^{208}+x^{176}+x^{112}+x \pmod{f(x)}$이고 272와 208, 176, 112 의 차는 각각 64, 32, 64로 모두 8의 배수이므로 워드 시프트 연산으로 처리 가능하다. x는 1-bit left shift로 처리 할 수 있

Table 1. Loop counter from 66 to 62 at Fast reduction algorithm over $GF(2^{271})$

Loop counter	Operation
i = 66	T = C〔66〕 C〔58〕 = C〔58〕⊕T C〔54〕 = C〔54〕⊕T C〔46〕 = C〔46〕⊕T C〔33〕 = C〔33〕⊕(T〉〉7) C〔32〕 = C〔32〕⊕(T〈〈1)
i = 65	T = C〔65〕 C〔57〕 = C〔57〕⊕T C〔53〕 = C〔53〕⊕T C〔45〕 = C〔45〕⊕T C〔32〕 = C〔32〕⊕(T〉〉7) C〔31〕 = C〔31〕⊕(T〈〈1)
i = 64	T = C〔64〕 C〔56〕 = C〔56〕⊕T C〔52〕 = C〔52〕⊕T C〔44〕 = C〔44〕⊕T C〔31〕 = C〔31〕⊕(T〉〉7) C〔30〕 = C〔30〕⊕(T〈〈1)
i = 63	T = C〔63〕 C〔55〕 = C〔55〕⊕T C〔51〕 = C〔51〕⊕T C〔43〕 = C〔43〕⊕T C〔30〕 = C〔30〕⊕(T〉〉7) C〔29〕 = C〔29〕⊕(T〈〈1)
i = 62	T = C〔62〕 C〔54〕 = C〔54〕⊕T C〔50〕 = C〔50〕⊕T C〔42〕 = C〔42〕⊕T C〔29〕 = C〔29〕⊕(T〉〉7) C〔28〕 = C〔28〕⊕(T〈〈1)

다. Table 2는 감산 연산 수행 과정에서 각 워드가 감산되어 최종적으로 이동하는 워드를 나타낸다. 즉, 감산된 1번째 워드 'C〔0〕'에는 'C〔34〕, C〔42〕,C〔46〕,C〔50〕,C〔54〕,C〔62〕' 워드가 각각 1-bit left shift 된 상태로 XOR 연산이 일어나게 된다. 또한 15번째 워드 'C〔14〕'에는 'C〔48〕, C〔56〕,C〔60〕,C〔64〕' 가 1-bit left shift 된 상태와 'C〔34〕,C〔42〕,C〔46〕,C〔50〕,C〔54〕,C〔62〕'가 XOR 연산이 일어난다.

미리 계산된 감산 테이블을 이용하면 효율적으로 워드단위 연산들을 조합하여 연산량을 줄일 수 있다. Table 2를 살펴보면 중복으로 나타나는 워드들의 조합들을 발견할 수 있다. 예를 들어, 14번째 워드

Table 2. Reduction algorithm table over $GF(2^{271})$

Word No.	1-bit left shift	Just moving
0	34,42,46,50,54,62	
1	35,43,47,51,55,63	
2	36,44,48,52,56,64	
3	37,45,49,53,57,65	
4	38,46,50,54,58,66	
5	39,47,51,55,59,67	
6	40,48,52,56,60	
7	41,49,53,57,61	
8	42,50,54,58,62	
9	43,51,55,59,63	
10	44,52,56,60,64	
11	45,53,57,61,65	
12	46,54,58,62,66	
13	47,55,59,63,67	
14	48,56,60,64	34,42,46,50,54,62
15	49,57,61,65	35,43,47,51,55,63
16	50,58,62,66	36,44,48,52,56,64
17	51,59,63,67	37,45,49,53,57,65
18	52,60,64	38,46,50,54,58,66
19	53,61,65	39,47,51,55,59,67
20	54,62,66	40,48,52,56,60
21	55,63,67	41,49,53,57,61
22	56,64	34,46,58
23	57,65	35,47,59
24	58,66	36,48,60
25	59,67	37,49,61
26	60	34,38,42,46,54
27	61	35,39,43,47,55
28	62	36,40,44,48,56
29	63	37,41,45,49,57
30	64	38,42,46,50,58
31	65	39,43,47,51,59
32	66	40,44,48,52,60
33	67	41,45,49,53,61

에 XOR 연산 될 'C〔34〕⊕C〔42〕⊕C〔46〕⊕C〔50〕⊕C〔54〕⊕C〔62〕'는 0번째에서 1-bit left shift 처리하여 재사용 할 수 있다. 그 이외에도 2개 혹은 3개씩 쌍으로 중복되는 부분들 중에 빈도수가 높은 것들부터 중복 사용하여 연산량을 줄일 수 있다. 그리고 한번 로드된 워드들은 다시 로드 하지 않기 위해 레지스터가 허락하는 한 값을 유지하면서 최대한 사용을 한다. 따라서 주어진 26개의 레지스

Algorithm 4. Fast reduction modulo $f(x)=x^{193}+x^{145}+x^{129}+x^{113}+1$ (with 8-bit processor)

Input: $c(x)$ which of degree is at most 384
Output: $c(x) \pmod{x^{193}+x^{145}+x^{129}+x^{113}+1}$

1: T ← C[48].
2: C[42] ← C[42]⊕T.
3: C[40] ← C[40]⊕T.
4: C[38] ← C[38]⊕T.
5: C[23] ← C[23]⊕(T<<7).
6: for i from 47 downto 25 do
  6.1: T ← C[i].
  6.2: C[i-6] ← C[i-6]⊕T.
  6.3: C[i-8] ← C[i-8]⊕T.
  6.4: C[i-10] ← C[i-10]⊕T.
  6.5: C[i-24] ← C[i-24]⊕(T>>1).
  6.6: C[i-25] ← C[i-25]⊕(T<<7).
7: T ← C[24]&0xFE.
8: C[18] ← C[18]⊕T.
9: C[16] ← C[16]⊕T.
10: C[14] ← C[14]⊕T.
11: C[0] ← C[0]⊕(T>>1).
12: C[24] ← C[24]&0x01.
13. Return (C[24],⋯,C[0])

터를 이용하여 중복되는 워드들의 조합들을 어떻게 효율적으로 스케줄링 하느냐가 XOR, LOAD, STORE 연산량 최소화에 대한 관건이다.

4.2 GF(2^{193}) 상에서의 감산 알고리즘

$GF(2^{271})$과 마찬가지로 $GF(2^{193})$의 원소는 8-bit 환경에서 25개의 워드로 구성이 된다. 유한체 감산 연산은 곱셈과 제곱에 대한 결과 49개의 워드를 입력받고 감산 연산의 결과로 25개의 워드를 출력한다. $GF(2^{193})$에서는 기약다항식을 $f(x)=x^{193}+x^{145}+x^{129}+x^{113}+1$으로 선택하였다. 그에 따른 Fast reduction 알고리즘은 Algorithm 4와 같다. 상수를 제외한 나머지 항들 간의 지수의 차가 8의 배수라는 점이 $GF(2^{271})$의 기약다항식과 같다. 따라서 워드 단위의 이동으로 쉽게 감산 연산이 구현 가능하다. 이 기약다항식의 또 다른 특징으로는 중간항($x^{145}, x^{129}, x^{113}$)들끼리의 지수의 차이가 16으로 상대적으로 작다. Table 3은 기약다항식 $f(x)=x^{193}+x^{145}+x^{129}+x^{113}+1$에 대한 Fast reductio-n의 반복 루프 47부터 43까지에 대한 동작을 기술한 것이다. 루프 카운터의 홀수부(....)와 짝수부(__)에서 각각 중복 로드와 저장이 발생하고 매 루프마다 시프트 연산이 발생하는 부분(___)에서도 중복이 발생한다. 결국 중간항들의 지수 차이가 16으로 작아져서 중복되는 메모리 로드와 저장이 $GF(2^{271})$의 경우보다 더 많이 일어나는 것을 알 수 있다. 감산 연산중에 'C[40],C[39]'는 2회, 'C[38]~C[24]'는 3회, 'C[23]~C[0]'는 5회 메모리 로드와 저장이 중복 수행 된다.

Table 3. Loop counter from 47 to 43 at Fast reduction algorithm over $GF(2^{193})$

Loop counter	Operation
i = 47	T = C[47] C[41] = C[41]⊕T C[39] = C[39]⊕T C[37] = C[37]⊕T C[23] = C[23]⊕(T>>1) C[22] = C[22]⊕(T<<7)
i = 46	T = C[46] C[40] = C[40]⊕T C[38] = C[38]⊕T C[36] = C[36]⊕T C[22] = C[22]⊕(T>>1) C[21] = C[21]⊕(T<<7)
i = 45	T = C[45] C[39] = C[39]⊕T C[37] = C[37]⊕T C[35] = C[35]⊕T C[21] = C[21]⊕(T>>1) C[20] = C[20]⊕(T<<7)
i = 44	T = C[44] C[38] = C[38]⊕T C[36] = C[36]⊕T C[34] = C[34]⊕T C[20] = C[20]⊕(T>>1) C[19] = C[19]⊕(T<<7)
i = 43	T = C[43] C[37] = C[37]⊕T C[35] = C[35]⊕T C[33] = C[33]⊕T C[19] = C[19]⊕(T>>1) C[18] = C[18]⊕(T<<7)

Table 4는 감산처리 될 상위 워드들을 최종적으로 감산시킨 형태로 테이블에 나타내었다. $GF(2^{271})$와 마찬가지로 어떻게 효율적으로 워드단위 연산들을

Table 4. Reduction algorithm table over $GF(2^{193})$

Word No.	1-bit right shift	Just moving
0	24,30,32,34,36,40,42	
1	25,31,33,35,37,41,43	
2	26,32,34,36,38,42,44	
3	27,33,35,37,39,43,45	
4	28,34,36,38,40,44,46	
5	29,35,37,39,41,45,47	
6	30,36,38,40,42,46,48	
7	31,37,39,41,43,47	
8	32,38,40,42,44,48	
9	33,39,41,43,45	
10	34,40,42,44,46	
11	35,41,43,45,47	
12	36,42,44,46,48	
13	37,43,45,47	
14	38,44,46,48	24,30,32,34,36,40,42
15	39,45,47	25,31,33,35,37,41,43
16	40,46,48	24,26,30,38,40,44
17	41,47	25,27,31,39,41,45
18	42,48	24,26,28,30,34,36,46
19	43	25,27,29,31,35,37,47
20	44	26,28,30,32,36,38,48
21	45	27,29,31,33,37,39
22	46	28,30,32,34,38,40
23	47,48	29,31,33,35,39,41
24		30,32,34,36,40,42

조합하여 스케줄링 하는가가 연산량을 좌우한다. 주어진 26개의 레지스터를 이용하여 중복되는 워드들의 조합들을 최대한 재사용하는 방식으로 연산량을 줄일 수가 있다. 예를 들어 Table 4에서 C[25]$\oplus$C[31]$\oplus$C[33]$\oplus$C[35]$\oplus$C[37]$\oplus$C[41]$\oplus$C[43]가 15번째 워드에서도 쓰이고 1번째 워드에서도 1-bit right shift하여 쓰이기 때문에 한번 계산으로 중복사용이 가능하다. 그 이외에도 2개 혹은 3개씩 쌍으로 중복되는 부분들 중에 빈도수가 높은것들부터 중복 사용하여 연산량을 최소화 할 수 있다.

V. 구 현

5.1 연산량 분석

Fig. 3과 Fig. 4에서는 감산 연산을 구현 시 실제 레지스터를 어떻게 사용하는지에 대한 스케줄링을 나타내고 있다. 가장 왼쪽 줄의 r0,r1...r25는 26개의 사용가능한 레지스터를 나타내고 각 행은 해당 레지스터에 저장되는 값을 나타낸다. 기본적으로 쉼표 (,)는 XOR연산을 나타내고, 다른 레지스터에 저장된 값의 복사를 나타내기 위해서는 굵은 네모칸으로 해당 값을 표시하였다. 이와 비슷하게 [41,49]와 같은 대괄호[]는 다른 레지스터에 있는 워드(41 XOR 49)의 값을 가져와 연산을 하겠다는 것을 나타낸다. (L=Load, 25*=rotate left 25, rol=rotate left, ror=rotate right, &=AND연산)

레지스터 스케줄링에서 [41,49,53,61],[40,48, 52,60],[39,47,51,59],[38,46,50,58]등과 같이 대괄호[]로 묶여 있는 워드 블록들을 많이 찾아 볼 수 있다. 이는 한번 계산으로 두 번 혹은 세 번씩 재사용을 한 감산 알고리즘 테이블의 중복 워드 조합들이 많다는 사실을 나타낸다. 기본적으로 25번째 레지스터(r25)는 다른 레지스터들(r0~r24)에서 이미 만들어진 워드들의 블록들을 이용하여 최종적인 결과값을 만들고 저장하는 공간으로 사용하였다.

Table 5. The number of operations for reduction algorithm

Operation	$GF(2^{271})$ Fast reduction	$GF(2^{271})$ Proposed reduction	$GF(2^{193})$ Fast reduction	$GF(2^{193})$ Proposed reduction
Load	208	94	146	50
Store	174	34	122	25
Shift	68	41	48	25
XOR	173	154	123	123

Table 5는 Fast reduction 알고리즘과 제안하는 알고리즘에서 필요한 주요 연산의 횟수를 비교한 표이다. 제안하는 감산 알고리즘의 연산량 측정을 위해 불필요한 연산과 레지스터 사용을 최대한으로 줄여서 최적화된 구현을 하였다. 따라서 Table 5에 나타난 연산량은 최적화를 했을 때의 값이다.

$GF(2^{271})$에서 메모리로부터 레지스터로 로드하는 연산은 Fast reduction 알고리즘이 208회가 필요하나 제안하는 감산 알고리즘은 94회로 약 2배 정도 차이가 났다. 메모리로 저장하는 연산은 각각 Fast reduction 174회, 제안하는 감산 알고리즘 34회로 약 5배 차이가 발생했다. 이는 Fast reduction의 루프 내에서 발생하는 메모리 중복 접근 때문이다. 최적화 구현에서 사용된 로드 연산은 'C[0]'부터 'C[41]'까지 1회, 'C[42]'부터 'C[67]'까지는 2회가 필요하며 연산이 끝난 후 메모리로 저장하는 연산은 각 워드의 연산이 종료된 후에만 저장하기 때문에 'C[0]'부터 'C[33]'까지 1회만 사용하게 된다. 동일한 이유로 $GF(2^{193})$에서 메모리로부터 레지스터로

r0	L45	45,[41,49,53,61],[67*]	L42	42 rol	42*,[50*],[54*],[58*],[62*]	
r1	L44	44,[40,48,52,60],[66*]	L43	43 rol	43*,[51*],[55*],[59*],[63*]	
r2	L43	43,[39,47,51,59],[65*]	L44	44 rol	44*,[52*],[56*],[60*],[64*]	
r3	L42	42,[38,46,50,58],[64*]	L45	45 rol	45*,[53*],[57*],[61*],[65*]	
r4	41	41,[37,45,49,57],[63*]	L46	46 rol	46*,[54*],[58*],[62*],[66*]	
r5	40	40,[36,44,48,56],[62*]	L47	47 rol	47*,[55*],[59*],[63*],[67*]	
r6	39	39,[35,43,47,55],[61*]				
r7	38	38,[34,42,46,54],[60*]				
r8	L61	61,[37,49],[59*],[67*]				
r9	L60	60,[36,48],[58*],[66*]				
r10	L59	59,[35,47],[57*],[65*]				
r11	L58	58,[34,46],[56*],[64*]				
r12	L41	41,49,53,61	41,49,53,57,61,[55*],[63*],[67*]			
r13	L40	40,48,52,60	40,48,52,56,60,[54*],[62*],[66*]			
r14	L39	39,47,51,59	39,47,51,55,59,67,[53*],[61*],[65*]			
r15	L38	38,46,50,58	38,46,50,54,58,66,[52*],[60*],[64*]			
r16	L37	37,49	37,45,49,57	37,45,49,53,57,65,[51*],[59*],[63*],[67*]		
r17	L36	36,48	36,44,48,56	36,44,48,52,56,64,[50*],[58*],[62*],[66*]		
r18	L35	35,47	35,43,47,55	35,43,47,51,55,63,[49*],[57*],[61*],[65*]		
r19	L34	34,46	34,42,46,54	34,42,46,50,54,62,[48*],[56*],[60*],[64*]		
r20	L46	L51	L56	L61	61 rol	
r21	L47	L52	L57			
r22	L48	L53	{[33,45,41,49,53,61,67*] & 0x80} rol	[34,42,46,50,54,62] rol	[35,43,47,51,55,63] rol	
r23	L49	L54	[36,44,48,52,56,64] rol	[37,45,49,53,57,65] rol	[38,46,50,54,58,66] rol	
r24	L50	L55	[39,47,51,55,59,67] rol	[40,48,52,56,60] rol	[41,49,53,57,61] rol	
r25	L62	62 rol	L63	63 rol	L64	64 rol
	L65	65 rol	L66	66 rol	L67	67 rol
	L33	(33,[45,41,49,53,61,67*]) & 0x7F	STORE	L32	32,[44,40,48,52,60,66*]	STORE
	L31	31,[43,39,47,51,59,65*]	STORE	L30	30,[42,38,46,50,58,64*]	STORE
	L29	29,[41,37,45,49,57,63*]	STORE	L28	28,[40,36,44,48,56,62*]	STORE
	L0	0,[(34,42,46,50,54,62)rol]	STORE	L1	1,[(35,43,47,51,55,63)rol]	STORE
	L2	2,[(36,44,48,52,56,64)rol]	STORE	L3	3,[(37,45,49,53,57,65)rol]	STORE
	L4	4,[(38,46,50,54,58,66)rol]	STORE	L5	5,[(39,47,51,55,59,67)rol]	STORE
	L6	6,[(40,48,52,56,60)rol]	STORE	L7	7,[(41,49,53,57,61)rol]	STORE
	L48	48 rol	L49	49 rol	L50	50 rol
	L51	51 rol	L52	52 rol	L53	53 rol
	L54	54 rol	L55	55 rol	L56	56 rol
	L57	57 rol	L58	58 rol	L59	59 rol
	L60	60 rol	L62	62 rol	L63	63 rol
	L64	64 rol	L65	65 rol	L66	66 rol
	L67	67 rol	L8	8,[(42,50,54,58,62)rol]	STORE	L9
	9,[(43,51,55,59,63)rol]	STORE	L10	10,[(44,52,56,60,64)rol]	STORE	L11
	11,[(45,53,57,61,65)rol]	STORE	L12	12,[(46,54,58,62,66)rol]	STORE	L13
	13,[(47,55,59,63,67)rol]	STORE	L14	14,[34,42,46,50,54,62,(48,56,60,64)rol]	STORE	L15
	15,[35,43,47,51,55,63,(49,57,61,65)rol]	STORE	L16	16,[36,44,48,52,56,64,(50,58,62,66)rol]	STORE	L17
	17,[37,45,49,53,57,65,(51,59,63,67)rol]	STORE	L18	18,[38,46,50,54,58,66,(52,60,64)rol]	STORE	L19
	19,[39,47,51,55,59,67,(53,61,65)rol]	STORE	L20	20,[40,48,52,56,60,(54,62,66)rol]	STORE	L21
	21,[41,49,53,57,61,(55,63,67)rol]	STORE	L22	22,[58,34,46,(56,64)rol]	STORE	L23
	23,[59,35,47,(57,65)rol]	STORE	L24	24,[60,36,48,(58,66)rol]	STORE	L25
	25,[61,37,49,(59,67)rol]	STORE	L26	26,[38,34,42,46,54,60*]	STORE	L27
	27,[39,35,43,47,55,61*]	STORE				

Fig. 3. Schedule for using resisters (reduction over $GF(2^{271})$)

로드는 Fast reduction 알고리즘은 146회가 발생하나 제안하는 알고리즘은 50회로 약 3배 차이가 났다. 레지스터에서 메모리로 저장은 122회에서 24회로 약 5배 차이가 났다. 최적화 구현에서 로드된 워드들은 'C[0]~C[48]'중에 'C[28]'을 제외하고는 모두 1회만 로드하였고 'C[28]'만 2번하였다. 메모리로 저장하는 연산은 'C[0]'부터 'C[24]'까지 1회만 수행하여 감산 연산을 종료하였다. Shift 연산은 $GF(2^{271})$ 에서 68회에서 41회로, $GF(2^{193})$ 에서는 48회에서 25회로 줄었다. 제안하는 감산 알고리즘은 미리 구해놓은 감산 알고리즘 테이블의 결과 값에 대해 순차적으로 Shift 연산을 진행 할 수 있으므로 Shift 연산의 연속성을 유지할 수 있다. 예를 들어, $GF(2^{271})$ 에서 Fast reduction은 하나의 워드 감산을 위해 7-bit right shift(>>7)와 1-bit left shift(<<1)가 필요하다. 하지만 제안하는 감산 알고리즘은 ATmega의 rotate left shift 명령어를 이용하여 Table 2의 0번째부터 33번째까지 1-bit left shift 연산을 순차적으로 처리할 수 있다. rotate left shift 명령어는 해당 워드를 1-bit left shift 하고 최상위 1-bit를 carry로 저장한다. 그리고 다음번에 rotate left shift 명령어가 있으면 carry값을 최하위 bit에 넣는다. 이를 통해 기존의 7-bit right shift(>>7)연산의 효과를 볼 수가 있다. 따라서 필요한 rotate 연산은 34번이지만 중간 값을 저장할 레지스터 개수가 제한적이라 여러 개의 블록으로 나누어 처리하여 총 41번의 rotate 연산으로 구현하였다. $GF(2^{193})$ 에서는 반대로 1-bit right shift를 최상위 워드부터 0번째까지 순차적으

r0	L 24	(24,[32,34,30,40],[36,42])&FE ror				
r1	L 30	30,40	26,(30,40,24)&FE,[38,48],[(32,34,36,42)&1]			
r2	L 32	32,34,[30,40]	32,34,30,40,[28,38]	45	43,45,[37,47] ror	
r3	L 34	(24,30,34,36)&FE,([32,40,42]&1),26,28,46	41	41,47 ror		
r4	L 36	36,42	36,42,44,[46,48&1] ror			
r5	L 40	40,[46,48&1] ror				
r6	L 42	42,[48&1] ror				
r7	30	26,30,32,36,48,[28,38]	L43			
r8	32	32,40,42	32,40,42,[48&1],[38,44] ror			
r9	34	34,40,42,46,44 ror				
r10	36	32,34,36,42	(32,34,36,42)&1	32,40,42	(32,40,42)&1	L38
	38,44	44	44 ror	L26	43	L25
	L27	27,43,[33,35,37],[39,45] ror				
r11	30,40	30,40,38,[36,42],[46,48&1] ror				
r12	32,34	32,34,[36,42]	32,34,36,42,26,[38,44] ror			
r13	34,36	28,34,36,40,46,[38,48] ror				
r14	(24,32,34,30,40,36,42)&1	STORE	48&1	46,48&1	46,48&1,[38,44] ror	
r15	L 29	29,35,41,[37,47],[39,45] ror				
r16	L 31	31,39	31,39,[37,47],[41,43] ror			
r17	L 33	33,35,37				
r18	L 35	35,4745,[41,43] ror				
r19	L 39	39,45				
r20	L 41	41,43				
r21	31	25,31	25,31,[33,35,37],[41,43] ror			
r22	33	33,[41,43],[39,45] ror				
r23	47	47,[39,45] ror				
r24	L48	(48&1) ror	L47	47 ror	L46	46 ror
	L28	28,38	L45	45ror	L27	L37
	37,47					
r25	L23	23,29,41,[31,39],[33,35],[47ror]	STORE	L22	22,[32,34,30,40,28,38],[46ror]	STORE
	L21	21,27,29,33,37,[31,39],[45ror]	STORE	L44	L20	20,[26,30,32,36,48,28,38],[44ror]
	STORE	L19	19,27,[25,31],[29,35],[37,47],[43ror]	STORE	L18	18,[(24,30,34,36)&FE,(32,40,42)&1,26,28,46] ,[42,(48&1)ror]
	STORE	L17	17,27,41,[25,31],[39,45],[41,47ror]	STORE	L16	16 ,[26,(30,40,24)&FE,38,48,(32,34,36,42)&1] ,[40,46,48&1 ror]
	STORE	L15	15 ,[25,31,33,35,37,41,43] ,[47,39,45 ror]	STORE	L14	14,[(24,32,34,30,40,36,42)&FE] ,[46,48&1,38,44 ror]
	STORE	L13	13,[43,45,37,47 ror]	STORE	L12	12,[36,42,44,46,48&1 ror]
	STORE	L11	11,[35,4745,41,43 ror]	STORE	L10	10,[34,40,42,46,44 ror]
	STORE	L9	9,[33,41,43,39,45 ror]	STORE	L8	8,[32,40,42,48&1,38,44 ror]
	STORE	L7	7,[31,39,37,47,41,43 ror]	STORE	L6	6,[30,40,38,36,42,46,48&1 ror]
	STORE	L5	5,[29,35,41,37,47,39,45 ror]	STORE	L4	4,[28,34,36,40,46,38,48 ror]
	STORE	L3	3,[27,43,33,35,37,39,45 ror]	STORE	L2	2,[32,34,36,42,26,38,44 ror]
	STORE	L1	1,[25,31,33,35,37,41,43 ror]	STORE	L0	0, [(24,32,34,30,40,36,42)&FE ror]
	STORE					

Fig. 4. Schedule for uising resisters (reduction over $GF(2^{193})$)

로 rotate right shift 명령어로 처리한다. rotate right shift 명령어는 해당 워드를 1-bit right shift 하고 최하위 1-bit를 carry로 저장한 후에 다음 rotate right shift 명령어가 있으면 carry값을 최하위 bit에 넣는다. 이때 7-bit left shift (<<<7)효과를 볼 수 있다. $GF(2^{193})$은 $GF(2^{271})$보다 상대적으로 레지스터의 여유가 있어서 처음부터 끝까지 순차적으로 rotate right shift 연산을 적용할 수 있다. 따라서 Shift 연산은 총 25번 소요가 되었다.

반면 Fast reduction 알고리즘에서는 각 루프마다 1-bit shift와 7-bit shift 연산이 적용된 값에 대해 각각 XOR 연산을 적용 시키므로 매 루프마다 2회의 Shift 연산이 필요하게 된다.

5.2 구현 결과

레지스터 내부의 데이터에 대하여 수행되는 연산에는 1클럭 사이클만이 소요되는데 반하여, 메모리에 저장된 데이터를 레지스터로 로드하거나 연산의 결과값을 다시 메모리에 저장하는 데는 2클럭 사이클이 소모된다. 따라서 감산 알고리즘에 메모리 로드와 저장을 최소한으로 설계하여 어셈블리 언어로 구현하였다.

Table 6는 기존의 Fast reduction과 본 논문에서 제안한 감산 연산 알고리즘을 각각 어셈블리 언어로 구현하였을 때의 성능을 클럭 사이클 단위로 비교 한 것이다. $GF(2^{271})$에서 기약다항식 $f(x)= x^{271}+x^{207}+x^{175}+x^{111}+1$에 대한 Fast reductio-n을 구현 시 1,084 클럭 사이클이 소모되었고, 제안하는 감산 연산 알고리즘을 사용하여 구현 시 482 클럭 사이클이 소모되어 약 56% 정도 속도가 향상되었다. $GF(2^{193})$에서 기약다항식 $f(x)= x^{193}+x^{145}+x^{129}+x^{113}+1$에 대한 Fast reduction은 769 클럭 사이클이 소모되었고, 제안하는 감산 연산 구현 시 326 클럭 사이클로 약 58% 가량 속도가 향상되었다. 단, 이 수치들은 오직 알고리즘 자체에 대한 향상율이다. 실제 공개키 암호시스템의 구현을 위해서는 부가적인 처리들이 필수적으로 추가된다. 예를 들어 입력 또는 출력 메모리의 주소 값이 들어있는 레지스터는 사용하기 전에 미리 push 명령어로 스택에 넣었다가 감산이 종료된 후에 다시 pop

명령어로 주소 값을 받아와야 다음 연산을 진행하는데 지장을 주지 않는다. 추가 연산으로 인해 $GF(2^{271})$ 에서 Fast reduction은 1,084 클럭 사이클에서 1,126 클럭 사이클로 늘어나고, 제안하는 알고리즘은 482 클럭 사이클에서 524 클럭 사이클로 늘어난다. $GF(2^{193})$ 에서는 Fast reduction이 769 클럭 사이클에서 811 클럭 사이클로 늘어나고, 제안하는 알고리즘은 326 클럭 사이클에서 368 클럭 사이클로 늘어난다. 추가된 클럭 사이클은 전체적인 향상율을 저하시키게 된다. 따라서 실질적인 감산 연산 향상율은 각각 약 53%, 55%로 나타났다.

또한 Table 5을 기준으로 연산량을 계산하면 Table 6의 연산량과 일치 하지 않음을 알 수 있다. 실제 알고리즘을 구현할 때 LOAD, STORE, SHIFT, XOR 연산들이 주를 이루는 것이 사실이지만 그 이외에도 구현을 용이하기 위한 연산들이 필요하다. 예를 들어 주소 참조 레지스터의 값을 수정하기 위해서 adiw, sbiw와 같은 명령어들이 필요하고, 필요한 상수값을 불러오기 위해 ldi 명령어가 필요하다. 그리고 구현 중에 발생하는 캐리 값을 지우거나 레지스터 값을 비우고 0으로 만들기 위해 clc, clr과 같은 명령어들도 필요하다. 이처럼 구현에서 필요한 부가적인 연산들로 인해 Table 5를 기준으로 한 연산량과 Table 6의 연산량 사이에서 차이가 발생한다.

Table 6. Comparison of efficiency for implementation result

Field		Fast reduction (clock cycles)	Proposed reduction (clock cycles)	Improvement
$GF(2^{271})$	Algorithm cost	1084	482	56%
	Implementation result	1126	524	53%
$GF(2^{193})$	Algorithm cost	769	326	58%
	Implementation result	811	368	55%

VI. 결 론

본 논문에서는 8-bit ATmega128 프로세서 환경에서 효율적인 $GF(2^{271})$, $GF(2^{193})$ 의 감산 알고리즘을 제안하였다. Fast reduction 알고리즘에서 반복적으로 일어나는 메모리의 호출과 저장으로 인한 비효율성을 개선하기 위해 〔3〕의 감산 알고리즘을 변형하여 사용하였다. 그러나 〔3〕의 감산 알고리즘과는 달리 감산 처리해야 될 상위 워드들을 미리 계산해서 중첩되는 워드들을 제거하였다. 결국 최종적인 감산 결과값을 제시하는 방식으로 불필요한 반복문 상의 중복 연산들을 제거하였다. 또한 최종적인 감산 결과 값들 사이에서는 워드들의 중복 조합들이 많이 존재한다. 이런 중복 조합들은 한번 계산해 놓으면 여러 번 중복 사용이 가능하여 레지스터 사용량과 연산량을 효율적으로 줄일 수가 있다. 따라서 최대한으로 Table에 나타난 중복 조합들을 사용할 수 있도록 스케줄링 하는 것이 구현에서의 관건이다. 본 논문에서는 제한된 환경에서 레지스터를 적극적으로 사용하기 위해 중복 조합들을 워드 2개 단위까지 재사용하여 $GF(2^{271})$, $GF(2^{193})$ 에서 각각 53%, 55%의 효율성을 개선 시켰다.

또한, ATmega128에서 제공하는 rotate right, rotate left 연산을 적극 활용하였다. Fast reduction 반복문에서의 1-bit shift와 7-bit shift를 한번의 rotate연산으로 처리함으로써 연산의 효율성을 높였다.

8-bit ATmega128 프로세서 기반으로 이진체 곱셈 연산에서 감산 연산이 차지하는 비중은 약 4% 미만으로 작다. 그러나 이진체 제곱 연산에서 감산 연산은 약 50% 정도로 차지하는 비중이 크다. 따라서 제안하는 감산 연산 알고리즘은 이진체 제곱 연산에서 큰 효율성을 얻을 수 있다.

자원이 제한된 환경에서는 중복되는 메모리 접근에 대한 부하가 상대적으로 크다. 미리 계산된 감산 연산 알고리즘 구성표를 통해서 레지스터 사용을 효율적인 설계가 가능하였다. 따라서 기존의 Fast reduction을 적용하였을 때 생겨나는 중복되는 메모리 접근을 효과적으로 줄일 수 있었다. 위와 같은 최적화 기법을 다른 기약다항식을 사용하는 이진체에서도 유도가 가능하다. 중간항들의 지수의 차이가 작고 8이나 16의 배수인 기약다항식이라면 더 좋은 향상율을 기대 할 수 있다. 또한 중간항들의 지수의 차가 16의 배수인 기약다항식에 대해서는 16-bit 프로세서 환경에서도 본 논문에서 제시한 감산 연산 알고리즘을 적용 할 수 있다. 이를 반영하여 다양한 이진체와 기약다항식의 특성을 고려한 효율적인 감산 알고리즘에 대한 연구를 진행 중이다.

References

〔1〕 N. Koblitz. "Elliptic curve cryptosystems", Mathematics of Computation, vol. 48, no. 177, pp. 203-209, Jan. 1987

〔2〕 V.S. Miller. "Use of elliptic curves in cryptography," Advances in Cryptology, CRYPTO'85, LNCS 218, pp. 417-426, 1986.

〔3〕 Seog Chung Seo, Dong-Guk Han and Seokhie Hong, "TinyECCK : Efficient Implementation of Elliptic Curve Cryptosystem over $GF(2^m)$ on 8-bit Micaz Mote," Journal of the Korea Institute of Information Security and Cryptology, 18(3), pp.1338-1347, 2008.

〔4〕 M. Scott, "Optimal irreducible polynomials for $GF(2^m)$ arithmetic," Cryptology ePrint Archive 2007-192, May. 2007.

〔5〕 ATmel ATmega128(L) Datasheet, http://www.atmel.com, 2006.

〔6〕 D. Hankerson, A. Menezes and S. Vanstone, Guide to elliptic curve cryptography, Springer-Verlag, pp. 53-56, 2004.

〈저자소개〉

박 동 원 (Dong-won Park) 학생회원
2013년 2월: 숭실대학교 수학과 학사
2013년 9월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 정보보호, 공개키 암호시스템

권 희 택 (Heetaek Kwon) 학생회원
2010년 2월: 고려대학교 수학과 학사
2010년 3월~현재: 고려대학교 정보보호대학원 석박사 통합과정
〈관심분야〉 정보보호, 공개키 암호시스템

홍 석 희 (Seok-hie Hong) 종신회원
1995년 2월: 고려대학교 수학과 학사
1997년 2월: 고려대학교 수학과 석사
2001년 8월: 고려대학교 수학과 박사
1999년 8월~2004년 2월: (주) 시큐리티 테크놀로지스 선임연구원
2003년 8월~2004년 2월: 고려대학교 정보보호기술연구센터 선임연구원
2004년 4월~2005년 2월: K.U.Leuven, ESAT/SCD-COSIC 박사후연구원
2005년 3월~2013년 8월: 고려대학교 정보보호대학원 부교수
2013년 9월~현재: 고려대학교 정보보호대학원 정교수
〈관심분야〉 대칭키·공개키 암호 분석 및 설계, 컴퓨터 포렌식