NDK 기반 공개키 암호를 위한 곱셈기 구현 및 분석

서화정* · 김호원**

Implementation and Analysis of Multi-precision Multiplication for Public Key Cryptography Based on NDK

Hwa-jeong Seo* · Ho-won Kim**

이 논문은 2010년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No.2010-0026621).

요 약

안드로이드 상에서의 프로그램 개발은 JAVA SDK를 통해 이루어진다. 하지만 JAVA의 사용은 virtual machine 상에서의 동작으로 인해 기존의 C언어에 비해 성능이 떨어지는 단점을 가진다. 최근에는 이러한 문제점을 해결하기 위해 NDK를 이용한 최적화 프로그램 개발이 사용되고 있다. 해당 기법은 안드로이드 프로그램을 C언어로 작성하여 연산을 효율적으로 수행한다. 본 논문에서는 안드로이드 상에서의 공개키 기반 암호화를 비교 분석하기 위해 NDK와 SDK를 사용하여 곱셈을 구현한다. SDK의 구현에는 BigInteger 패키지를 사용하였으며 NDK의 구현에는 Comb method를 사용하였다. 또한 안드로이드 상에서의 사칙연산, 조건문 그리고 호출문의 연산 수행 결과를 비교하여 NDK를 통한 성능향상에 대해 알아본다.

ABSTRACT

On Android environment, program development is conducted with JAVA SDK. However, using JAVA, it is operated over virtual machine which shows lower performance in terms of speed than traditional C language programming. The method writes program in C language, which conducts operation efficiently. In the paper, we implement multiplication using NDK and SDK to analyze the public key cryptography over Android environment. In case of SDK, we used BigInteger package and in case of NDK, we used Comb method. Moreover, execution time of arithmetic, branch and call operations over Android environment is compared to understand performance enhancement using NDK package.

키워드

NDK, 안드로이드, 공개키 암호화, 곱셈기법

Key word

NDK, Android, Asymmetric Cryptography, Multiplication

* 정회원 : 부산대학교 컴퓨터공학과 박사과정 (hwajeong@pusan.ac.kr)
** 종신회원 : 부산대학교 컴퓨터공학과 교수

접수일자 : 2012. 06. 07
심사완료일자 : 2012. 08. 07

Open Access http://dx.doi.org/10.6109/jkiice.2012.16.11.2347

Ⅰ. 서 론

구글에서 개발한 안드로이드 플랫폼은 높은 완성도와 공개성으로 인해 가장 널리 사용되는 플랫폼이다[4, 14]. 현재 안드로이드 운영체제를 사용하는 스마트폰을 통한 다양한 서비스의 제공이 본격화되고 있다. 해당 서비스는 기본적인 게임, 유틸리티 그리고 인터넷 뱅킹까지 그 범위가 점차 확대되어 가고 있는 추세이다. 따라서 이전에는 심각하게 고려되지 않았던 스마트폰 상에서의 보안문제가 유용한 서비스제공을 위해 우선시 되어야 하는 선결 조건으로 여겨지고 있다.

기본적으로 스마트폰 상에서는 API로 제공하는 다양한 암호화 모듈을 통해 사용자가 전송하는 정보를 안전하게 보호하는 것이 가능하다. 하지만 자바의 가상머신 위에서 동작하는 API는 불필요한 코드를 많이 포함하게 되고 결과적으로 많은 Clock cycle을 소모하게 된다.

스마트폰과 같이 자원한정적인 장비에서는 최적화된 암호화모듈 구현을 통해 암호화과정을 단축시키고 스마트폰을 최대한 절전모드로 유지하는 것이 그 무엇보다 중요하다. 최적화된 암호화 모듈 구현을 위해서는 기계어에 가까운 하위 언어를 사용하여 구현하는 것이 보다 효율적이다.

지금까지 프로그램 작성 시 C와 C++언어는 장비를 효율적으로 동작시키기 위한 언어로 가장 많이 사용되어 왔다. 하지만 안드로이드 플랫폼에서는 안드로이드 SDK에서 제공하는 JAVA언어를 통해 구현하여야 한다[5].

안드로이드는 자바 가상머신 위에서 동작하고 있지만 그 하위 단은 여전히 C언어를 통해 동작하고 있음을 알 수 있다. 이를 탄력적으로 이용하기 위해 2009년 6월에는 안드로이드 NDK가 구글에 의해 공개되었다[6, 15]. 이는 장비를 조작하기 위한 framework단의 한계를 넘어 C언어로 안드로이드를 동작시키는 것이 가능하게 하여 기존의 자바기반의 프로그램 구현에 비해 속도가 개선되는 효율성을 가진다. 따라서 사용자가 이를 효과적으로 사용한다면 프로그램의 성능 향상이 가능하다.

본 논문에서는 NDK를 통한 공개키 기반 암호화를 위한 곱셈기를 NDK를 통해 구현한다. 이를 통해 ECC와 RSA와 같은 공개키 암호화를 안드로이드 상에서 구현하기 위한 방안을 제시한다.

본 논문의 구성은 다음과 같다. 2장에서는 현재 안드로이드의 전체적인 동작방식과 NDK를 통한 개발론을 제시하며, 기본적인 NDK와 자바상에서의 프로그램 성능을 비교해 본다. 3장에서는 본 논문에서 비교하게 될 API와 곱셈 구현기법을 살펴본다. 4장에서는 두 모듈에 대한 성능을 분석하며 마지막으로 5장에서는 본 논문에 대한 결론을 내린다.

Ⅱ. 관련연구

2.1. JNI[1]

안드로이드 프레임 워크에서 C/C++과 자바와 유기적으로 연결되기 위해서는 상위계층인 자바와 하위계층인 C/C++레이어를 상호 연결해 주는 매개체가 필요하다[7, 8].

[그림 1]과 같이 안드로이드에서 자바와 C/C++ 모듈간의 인터페이스가 가능하게 해주는 것은 JNI(Java Native Interface)이다. JNI는 다음과 같은 기능을 활용하기 위해 많이 사용되고 있다.

◇ 빠른 처리속도를 요하는 프로그램

JNI를 통한 C/C++언어를 통한 프로그램 작성은 자바를 통한 프로그램 작성에 비해 최적화된 코드를 제공함으로써 일반적으로 자바에 비해 높은 성능을 보인다. 따라서 성능이 중요한 프로그램의 경우 JNI를 통해 C/C++을 직접 동작시킬 수 있도록 한다.

◇ 하드웨어 제어

하드웨어 제어 코드는 C언어상에서 작성이 가능한데 이를 JNI를 통해 자바에서도 조작이 가능하게 된다.

◇ 프로그램 재사용

기존에 C/C++로 제작되어 사용되고 있는 라이브러리와 프로그램의 경우 자바에서 사용하기 위해서는 JNI를 통해 C/C++에 대한 인터페이스를 제공받아 활용하는 것이 가능하다.

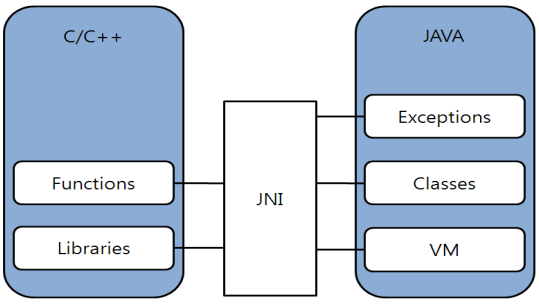

그림 1. 자바와 C언어를 연결하는 매개체
Fig. 1 Medium for connecting between JAVA and C language

하지만 JNI의 최적화된 코드를 사용하여 작성된 프로그램은 다른 프로그램과의 호환성이 떨어지며 안전도가 낮아지는 단점도 지닌다[9].

2.2. NDK와 SDK에 대한 비교

안드로이드 상에서의 프로그램의 구현은 현재 자바를 이용하여 달빗 가상머신 상에서 구현된다[6]. 하지만 성능상의 이유로 기존의 SDK방식이 아닌 NDK를 이용한 프로그램 작성이 본격화되고 있다. NDK를 활용한 구현 기법은 물리 시뮬레이션과 신호처리와 같은 영역에 보다 효과적이다. 또한 기본에 작성된 C/C++언어를 재사용하는 것이 가능하다는 장점을 가진다. [그림 2]는 자바를 통해 작성된 프로그램을 나타낸다. 여기서는 z=((z+1)*x)-y연산을 1000000번 수행하며 자바 상에서의 명령어 수행속도를 측정해 보았다. 사칙 연산을 복잡하게 구성한 이후는 C언어와 자바에서 가지는 코드 최적화로 인해 생기는 차이를 최소화하기 위해서이다.

```
z=0;
for(int kk=0;kk<1000000;kk++){

z = ((z+1)*x)-y;
}
```

그림 2. 자바를 이용한 프로그램 작성
Fig. 2 Writing program in JAVA

[그림 3]은 C언어를 통해 동일한 프로그램을 작성한 것을 의미한다. 해당 연산은 1000000번 반복되어 수행되며 이때 수행된 시간은 밀리초 단위로 계산되어 나타내도록 하였다. 두 프로그램은 연산이 수행되는 환경이 각각 C언어와 자바로써 상이하지만 모두 동일한 연산을 수행하는 프로그램이다.

```
#include "first.h"

int first(int x, int y)
{
        int c,d;
        c=0;
        for(d=0;d<1000000;d++){
                c = ((c+1)*x)-y;
        }
        return c;
}
```

그림 3. C언어를 통한 프로그램 작성
Fig. 3 Writing program in C language

[그림 4]는 해당 연산이 수행되고 난 뒤에 소비된 시간을 비교한 화면이다. 그림에서 보는 바와 같이 C언어를 통한 프로그램 작성 시 약 10배 가까이 성능이 향상됨을 알 수 있다. 따라서 성능이 중요시 되는 프로그램의 경우에는 C와 같은 하위 계층을 이용하여 프로그램 작성이 용이함을 본 실험 결과를 통해 알 수 있다.

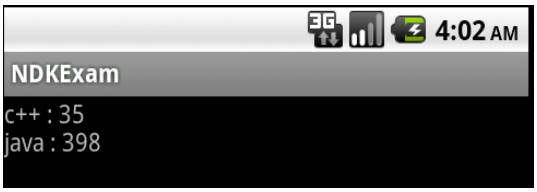

그림 4. 언어에 따른 수행시간 비교
Fig. 4 Comparison of execution time on language

2.3. Android BigInteger API[2]

안드로이드에서는 기본적으로 java.math.* 하위 라이브러리로 Big Integer구현에 대한 패키지를 포함한다. 따라서 사용자는 쉽게 해당 라이브러리를 포함하여 공개키 기반 암호화 프로그램 작성이 가능하다. [그림 5]는 안드로이드 개발자 모임의 메인화면으로써 사용자는 편리하게 자신이 원하는 라이브러리를 찾아서 사용하는 것이 가능하다. 우리는 여기서 Big Integer를

생성하는 부분과 해당 Integer를 곱셈하는 부분을 사용하게 된다.

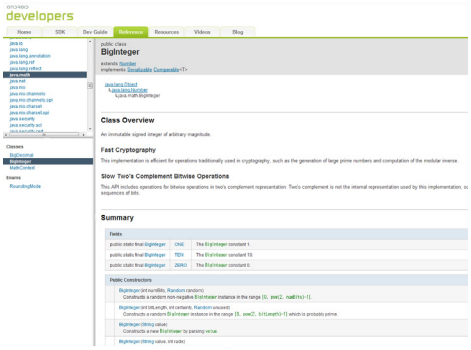

그림 5. Android Cryptography API
Fig. 5 Android Cryptography API

2.4. 이전연구 결과

2.4.1. Leonid Batyuk et al[11].

해당 논문에서는 안드로이드 환경과 자바 환경과의 성능의 차이점을 JRE 1.6을 이용하여 평가하였다. sorting알고리즘에 대한 성능을 JNI와 native code와의 조합을 통해 나타내었다. 안드로이드 환경 상에서는 JNI와 Native code를 사용하는 것이 Dalvik virtual machine만을 사용하는 것보다 성능이 좋게 나왔다. 하지만 Sun JRE의 경우에는 JNI와 Native code를 함께 쓰는 것이 JVM만을 사용하는 것보다 성능이 좋게 나오지는 않았다.

2.4.2. Lee, Jeon et al[12].

해당 논문에서는 JNI를 사용할 때 발생하는 지연, 정수 계산, 실수 계산, 메모리 접근 알고리즘 그리고 heap memory allocation 알고리즘에 대한 실험을 수행하였다. 실험 결과 JNI를 사용할 때 발생하는 지연은 다른 연산들의 계산 시 발생하는 이득으로 인해 상쇄되며 나머지 실험에서도 모두 NDK를 사용하여 구현한 결과가 성능이 높게 평가되었다. 특히 메모리와 heap에 대한 성능에서는 매우 큰 성능의 차이를 보였다.

2.4.3. Paik, Lee et al[13].

해당 논문에서는 JAVA를 통해 구현한 암호화 기법과 NDK를 사용했을 때의 성능을 비교 평가하고 있다. 또한 NDK를 보다 효율적으로 사용하기 위한 Framework을 제안하여 보다 효율적으로 NDK를 활용할 수 있도록 하고 있다.

Ⅲ. 안드로이드 상에서의 Big Integer 연산 모듈 구현

본 장에서는 공개키 기반 암호화의 기본 연산인 Big Integer연산을 보다 효율적으로 구현하기 위한 기법을 확인한다. 해당 연산은 공개키 기반 암호화를 위해 요구되는 체연산에 필수적이지만 임베디드 시스템의 bit의 크기가 정의된 체에 비해 작기 때문에 구현이 어렵다. 따라서 이를 구현하기 위한 방법론도 생각되어야 한다. 본 논문에서는 NDK를 통한 곱셈 모듈 설계를 안드로이드 상에 적합하게 수정하여 제시한다. 본 논문에서는 160-bit의 multi-precision 곱셈을 Comb method를 통해 구현한다. 이는 추후에 확대될 보다 다양한 체 상에서의 공개키 기반의 암호화와 타원곡선암호화까지 다양한 고급 암호화 구현에 효율적으로 활용이 가능하다.

```
Random a = new Random(1234);
BigInteger number = new BigInteger(160,a);
a.nextLong();
BigInteger number2 = new BigInteger(160,a);


long s1=System.nanoTime();
number2=number2.multiply(number);
long s2=System.nanoTime()-s1;
```

그림 6. Android API를 통한 곱셈 구현
Fig. 6 Multi-precision multiplication using Android API

3.1. 안드로이드 API를 통한 구현

안드로이드에서 제공하는 API를 동작시키기 위해 [그림 6]과 같이 암호화 과정을 수행하였다. 안드로이드 API에서는 BigInteger 패키지를 통해 사용자가 쉽게 Big Integer를 생성할 수 있다. 제시된 소스에서는 곱셈 수행시간을 측정하도록 작성되어 있다. 보다 정확한 측정을 위해 수행시간은 Big Integer를 생성하는 시간을 포함하지 않았다. 그 이유는 해당 연산은 인자를 생성하는 과정이므로 곱셈보다는 초기화 과정에 가깝기

때문이다. BigInteger 생성자에서는 곱셈에 사용되게 될 인자들을 사용되는 값에 따라 초기화하여 사용하게 된다. 곱셈은 두 개의 160-bit integer를 인자로 하여 연산을 수행하며 해당 연산의 앞뒤에서 수행시간을 측정하게 된다.

3.2. 제안하는 방식에 따른 구현

본 논문에서 C언어를 통한 곱셈의 구현에 사용한 곱셈 기법은 [3]에서 제안된 Comba 곱셈으로써 동일한 열에 위치한 값들을 지속적으로 축적하며 마지막에 한번만 해당 값을 저장하는 방식이다. 따라서 중간값을 저장하기 위한 메모리에 대한 접근이 줄어드는 장점을 가진다. 해당 곱셈기법을 계산하기 위해 본 논문에서는 64-bit long type을 선언해서 연산에 사용하였다. 하지만 java에서는 기존에 사용되었던 unsigned type이 가지는 문제점을 최소화하기 위해 해당 정의를 사용하지 못하도록 하고 있다.

따라서 안드로이드 상에서의 곱셈 구현기법은 64-bit long type을 사용하지만 실제 곱셈은 32-bit 미만의 값으로 곱셈을 취해야 한다. 그 이유는 32-bit이상의 값을 서로 곱해준 결과 값은 64번째 bit까지 사용하게 된다. 하지만 signed 형식에서는 64번째 bit는 부호 bit로써 연산의 결과를 저장할 수 없다. 따라서 마지막 64번째 bit는 사용하지 않는 범위 내에서 곱셈을 수행하기 위해 64-bit long 형식 안에서 29-bit만을 사용하여 곱셈을 구현하였다. 따라서 160-bit 곱셈을 수행하기 위해 인자를 29-bit씩 여섯 개씩 나누어 계산하였으며 자세한 구현은 [그림 8]과 같다.

```
long[] x = {0x1fffffffl,0x1fffffffl,0x1fffffffl,0x1fffffffl,0x1fffffffl,0x1fffffffl};
long[] y = {0x1fffffffl,0x1fffffffl,0x1fffffffl,0x1fffffffl,0x1fffffffl,0x1fffffffl};
long[] z={1,2,3,4,5,1,2,3,4,5,1,2};

long s3=System.nanoTime();
    add(x,y,z);
long s4=System.nanoTime()-s3;
```

그림 7. 자바 상에서의 c언어로 곱셈구현 연동
Fig. 7 Operating the multiplication over Java using C language

[그림 8]은 본 논문에서 사용한 곱셈이 C언어에서 호출된 부분을 나타낸다. 해당 함수는 JNI를 통해 자바와 연결되며 [그림 7]의 add함수가 C언어의 곱셈함수와 연결되어 유기적으로 곱셈명령을 수행하게 된다.

```
numbers=(*env)->GetLongArrayElements(e
numbers2=(*env)->GetLongArrayElements(
numbers3=(*env)->GetLongArrayElements(

numbers3[0]=numbers[0]*numbers2[0];
numbers3[1]=numbers3[0]>>32;

numbers3[1]+=numbers[0]*numbers2[1];
numbers3[1]+=numbers[1]*numbers2[0];
numbers3[2]=numbers3[1]>>32;

numbers3[2]+=numbers[0]*numbers2[2];
numbers3[2]+=numbers[1]*numbers2[1];
numbers3[2]+=numbers[2]*numbers2[0];
numbers3[3]=numbers3[2]>>32;

numbers3[3]+=numbers[0]*numbers2[3];
numbers3[3]+=numbers[1]*numbers2[2];
numbers3[3]+=numbers[2]*numbers2[1];
numbers3[3]+=numbers[3]*numbers2[0];
numbers3[4]=numbers3[3]>>32;

numbers3[4]+=numbers[0]*numbers2[4];
numbers3[4]+=numbers[1]*numbers2[3];
numbers3[4]+=numbers[2]*numbers2[2];
numbers3[4]+=numbers[3]*numbers2[1];
numbers3[4]+=numbers[4]*numbers2[0];
numbers3[5]=numbers3[4]>>32;
```

그림 8. 곱셈구현
Fig. 8 Implementation of Multiplication

즉 자바코드로 작성된 부분인 [그림 7]의 add함수가 수행되게 되면 JNI를 통해 이와 연관된 C언어 구현부분인 [그림 8]의 곱셈을 수행하게 된다. 곱셈의 구현은 Comba 기법에 따라 동일한 열에 위치한 값을 축적시키며 연산을 하도록 구현되었다. 값의 축적이 끝나고 옆의 열로 이동하는 경우에는 shift연산을 통해서 상위 단의 값을 32-bit 오른쪽으로 이동시켜 지속적인 축적이 가능하도록 하였다.

Ⅳ. 암호화 모듈의 성능 분석

4.1. NDK를 이용하여 구현한 곱셈 모듈의 성능 분석

본 장에서는 기존의 API와 NDK를 사용한 최적화 곱셈 구현의 성능을 확인해보도록 한다. 그 결과는 [그림 9]과 같다. 그림에서 보는바와 같이 현재 안드로이드에서 제공하는 API는 NDK에 비해 성능이 저하됨을 나타낸다. 그 이유는 자바를 통한 구현에서는 virtual machine 위에서 함수 호출이 이루어지기 때문에 높은 부하가 부과되기 때문이다. 따라서 NDK를 통한 구현은 자바를 통한 구현에 비해 최적화 구현이 가능하다. 따라서 빠른 연

산이 필요한 경우에는 NDK를 통해 최적화된 연산의 구현이 중요하다.

그림 9. 구현 결과 속도 비교
Fig. 9 Comparison of performance

속도와 함께 고려해야 할 부분은 바로 사용되는 메모리양이다. [그림 10]은 C언어와 자바 구현 시 생성되는 apk파일의 크기를 비교한 결과를 나타낸다. [그림 10]에서 왼쪽의 6.apk는 자바 API를 통해 생성된 파일로써 13.4 KB의 용량을 나타낸다. 오른쪽의 5.apk는 C언어를 통해 생성된 파일로써 93.5KB를 나타낸다. 즉 구현 속도를 향상시키기 위해 하위단의 C언어를 사용하여 최적화된 코드로 작성하게 되면 속도는 향상되지만 그만큼 코드의 양이 많아지기 때문에 패키지의 크기가 증가하는 단점을 가진다.

반면에 자바로 작성된 코드는 고급언어로 축약된 형태로 명령이 수행되기 때문에 작은 용량을 나타낸다. 결론적으로 [그림 9]과 [그림 10]를 토대로 생각해 볼 때 속도와 메모리 크기의 적당한 Trade-off를 통해 상황에 맞는 적합한 구현(속도, 메모리)을 찾아나가는 것이 중요하다.

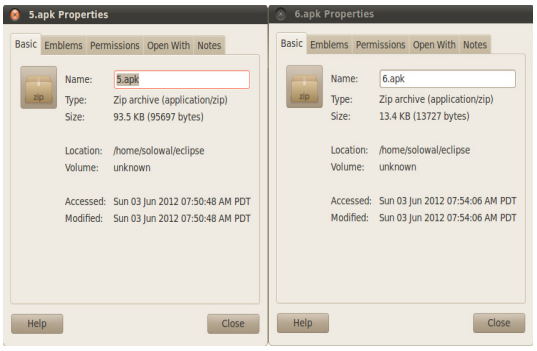

그림 10. 메모리 사용량 비교
Fig. 10 Comparison of memory consumption

4.2. NDK를 이용한 기본 연산 모듈의 성능평가

본 장에서는 기존의 논문들에서 다루지 않은 NDK를 사용함으로써 성능이 향상되는 연산들을 비교한다. 특히 NDK를 통한 암호화 모듈관점에서 자주 사용되는 연산들인 함수호출, 비트연산과 조건문을 분석함으로써 성능향상의 조건을 알아 볼 수 있도록 한다.

4.2.1. Function Call

NDK를 사용하기 위해서는 JAVA언어에서 NDK를 Call하기 위한 과정을 거쳐야 한다. 해당 연산에 대한 수행 횟수를 늘려가며 수행해 본 결과 10,000,000번의 연산 시에는 12737(msec)가 걸리며 상당히 많은 연산이 수행된다는 것을 알았다. 따라서 연산 속도 측정 시 NDK 안에서 연산이 지속적으로 수행되는 경우와 JAVA단에서 지속적으로 NDK에 접근하는 방식, 그리고 JAVA상에서 구현된 결과를 비교해 보았다.

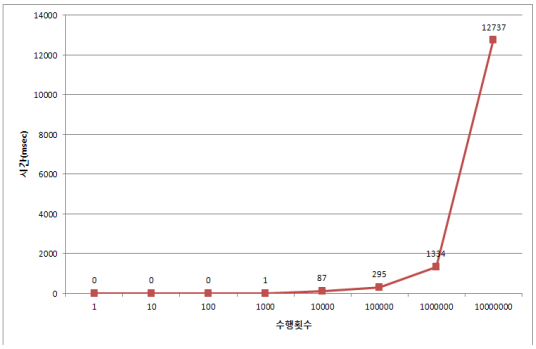

그림 11. Function Call
Fig. 11 Function Call

4.2.2. 비트연산

대칭키 암호화 과정에서 가장 많은 비중을 차지하는 연산은 비트연산이다. 본 장에서는 비트연산 시 소모되는 연산의 오버헤드에 대해 알아보도록 하겠다.

4.2.2.1. AND연산

AND연산은 암호화 과정에서 자주 사용되는 연산 중 하나이다. NDK에서 Function call을 사용한 경우 성능이 급격히 떨어졌지만 NDK안에서 연산을 반복하여 수행할 때는 JAVA에서 수행하는 경우에 비해 성능이 좋게 측정되었다.

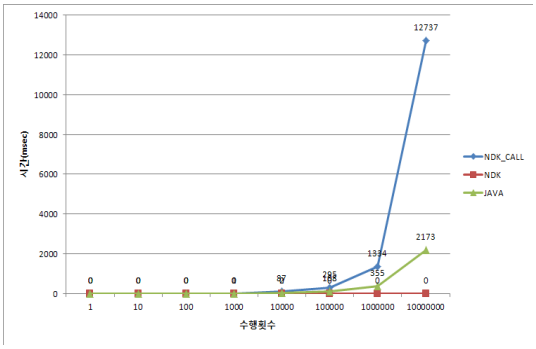

그림 12. AND 연산 비교
Fig. 12 Comparison of AND operation

4.2.2.2. Shift연산

Shift연산은 인자의 비트값을 정해진 수만큼 이동시키는 연산이다. 이동시키고자하는 비트의 수에 따라 성능이 결정되기 때문에 and연산과 같은 단순한 연산에 비해서는 요구되는 연산량이 높게 측정되었다. 여기서도 NDK단에서 반복적으로 수행한 경우에는 높은 성능을 나타내었다.

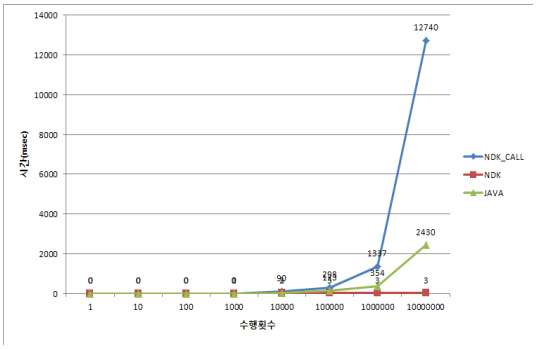

그림 13. Shift 연산 비교
Fig. 13 Comparison of shift operation

4.2.3. 조건문

조건문의 경우 NDK의 경우에는 shift연산과 비슷한 계산 량이 요구되었다. 하지만 JAVA의 경우에는 이전의 비트연산들에 비해 훨씬 높은 계산 량이 소모됨을 확인할 수 있었다.

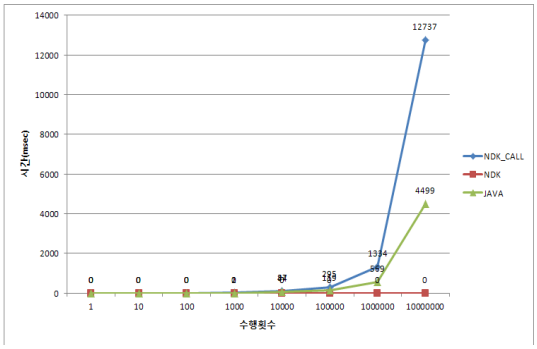

그림 14. 조건문 연산 비교
Fig. 14 Comparison of branch statement

V. 결 론

본 논문에서는 안드로이드 폰 상에서의 공개키 기반 암호화를 위한 최적화 곱셈 구현기법에 대해 알아보았다. 해당 기법은 NDK를 통한 C언어 프로그래밍을 통해 연산 수행 속도를 향상시켰다. C언어를 통해 기존의 library 기법보다 NDK 구현이 효율적임을 제시하였다. 이는 암호화와 그래픽 프로세싱과 같은 높은 성능이 요구되는 프로그램에 적용이 가능하다. 또한 function call에 소모되는 계산 량이 많음을 제시함으로써 NDK 사용 시 function call을 최소화해야 함을 보였다. 앞으로의 연구 방향은 본 논문에서 제시한 구현기법을 통해 공개키 알고리즘을 효율적으로 구현하여 안드로이드 운영체제에서 동작하는 암호화 통합 패키지를 제작하는 데 있다.

참고문헌

[1] http://developer.android.com/index.html, 안드로이드 개발관련 사이트
[2] http://developer.android.com/reference/java/math/BigInteger.html.
[3] Comba, P.: Exponentiation cryptosystems on the IBM PC. In: IBM Systems Journal 29(4), pp. 526-538. (1990)
[4] "Android.com," Available: http://www.android.com

[5] "Android SDK | Android Developers," Available: http://developer.android.com/sdk/index.html
[6] DalvikVM.com, "Dalvik Virtual Machine insights," Available: http://www.dalvikvm.com/
[7] "Android NDK | Android Developers," Available: http://developer.android.com/sdk/ndk/index.html
[8] "Java Native Interface - Wikipedia," Available: http://en.wikipedia.org/wiki/Java_Native_Interface
[9] Rob Gordon, "Essential JNI: Java Native Interface," ISBN 978-0136798958, 1998.
[10] Mark Allen Weiss, "C++ for Java Programmers," ISBN 978-0139194245, October 2003.
[11] Leonid Batyuk, Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Ahmet Camtepe and Sahin Albayrak, "Developing and Benchmarking Native Linux Applications on Android,"Proceedings of the 2nd International Conference on Mobile Wireless Middleware, Operating Systems, and Applications (Mobilware 2009), pp. 381-390, Beriln, Germany, April 28-29, 2009.
[12] Sangchul Lee and Jae Wook Jeon, "Evaluating Performance of Android Platform Using Native C for Embedded Systems," International Conference on Control, Automation and Systems, Gyeonggi-do, Korea, Oct. 27-30, 2010.
[13] Jung Ha Paik, Seog Chung SEO, Yungyu Kim, HwanJin Lee, HyunChul Jung, DongHoon Lee, "An Efficient Implementation of Block Cipher in Android Platform," 2011 Fifth FTRA International Conference on Multimedia and Ubiquitous Engineering. 2011.
[14] "What is android." [Online]. Available: http://developer.android.com/guide/basics/what-is-android.html
[15] "What is ndk." [Online]. Available: http://developer.android. com/sdk/ndk/overview.html

저자소개

서화정(Hwa-jeong Seo)

2004.3～2010.2 : 부산대학교 정보컴퓨터공학과 학사
2010.3～2012.2 : 부산대학교 컴퓨터공학부 석사
2012.3～현재 : 부산대학교 컴퓨터공학부 박사
※관심분야 : 정보보안, RFID/USN, 암호 이론, VLSI 설계

김호원(Ho-won Kim)

1989. 3～1993. 2 : 경북대학교 전자공학과 학사
1993. 3～1995. 2 : 포항공과대학교 전자전기공학과 공학석사
1995. 2～1999. 2 : 포항공과대학교 전자전기공학과 공학박사
1998.12～2008.2 : 한국전자통신연구원(ETRI) 정보보호연구단 선임연구원 / 팀장
2008. 3～현재 : 부산대학교 정보컴퓨터공학부 부교수
※관심분야 : 스마트그리드 보안, RFID/USN 정보보호 기술, PKC 암호, VLSI 설계, embedded system 보안