확인서 이용 부가형 디지탈서명 방식 표준(안)

임 채훈* 이 필중† 강 신각‡ 박성준§

요 약

정보화 사회에서 거의 모든 거래는 통신망을 통해 이루어지고, 이에 따라 통신망 상에서의 정보보호가 중요한 과제로 부각되고 있다. 이러한 통신망 상의 정보보호를 위해 필수적인 암호학적 도구가 디지탈서명이며, 국내에서도 고유의 디지탈서명 알고리듬을 표준화하려는 노력이 경주되어 왔다. 본 논문에서는 우리나라의 디지탈서명 표준안으로 상정된 디지탈서명 알고리듬 및 관련 시스템 변수의 선정 방법들을 기술한다. 또한 서명 표준안의 안전성, 효율성 등을 분석해 보고, 실제 구현결과도 제시한다.

제 1 절 서 론

정보화사회의 거의 모든 정보는 통신망을 통해 교환된다. 이에 따라 메시지의 수신자는 받은 메시지가 전송도중 내용이 바뀌지 않았는지, 또는 제 3자가 송신자를 가장하여 메시지를 보내지 않았는지 등을 확인할 수 있기를 바란다. 이런 목적으로 사용되는 가장 효율적인 도구가 디지탈서명이다.

디지탈서명은 서명자만이 알고있는 비밀키(이후 비공개 서명키 혹은 서명키로 부름)와 메세지의 함수로 계산되며, 서명자의 공개키(이후 공개 검증키 혹은 검증키로 부름)를 알면 누구나 사실 여부를 검증할 수 있으므로 디지탈 사회에서 발생할 수 있는 각종 보안문제를 해결하는데 매우 유용하게 쓰일 수 있다. 즉 메세지에 대한 디지탈서명은 누구나 그 메세지의 출처(작성자)를 확인할 수 있게 해 주는 인증 서비스, 획득한 메세지가 도중에 변경되었는 지의 여부를 알 수 있게 해 주는 무결성 서비스 및 메세지에 서명을 했다는 사실을 부인할 수 없게 하는 부인봉쇄 서비스 등 정보화사회에서 필요한 다양한 보안 서비스를 제공하는데 필수적인 도구이다.

현재 가장 널리 사용되는 디지탈서명 방식은 RSA 방식[28]과 ElGamal 방식[5]의 변형들이다. 이 두 방식은 안전성의 근거가되는 수학적 문제가 서로 다를 뿐만 아니라 서명 생성이나 검증 등 여러면에서 각기 장단점을 지니고 있다. RSA는 두 큰 소수의 곱인 합성수를 소인수분해하는 것이 어렵다는 사실을 이용한 서명 방식으로, 서명의 생성은 상당히 많은 계산량이 필요하나 검증이 매우 빠르게 수행될 수 있다. 현재 산업계에서 사용되는 대부분의 디지탈서명은 RSA를 기반으로 하고 있다. 특히 금융계통의 응용에서는 키분배나 디지탈서명용으로 거의 전적으로 RSA가 사용되고 있는 실정이다.

그러나 RSA는 암/복호화와 서명의 생성/검증 사이에 서로 역관계가 성립되어 서명방식이 암/복호화용으로 전용될 수 있다는 우려 때문에 비밀성을 배제하고 순수하게 서명만이 요구되는 경우 국가기관의 통제에 민감한 경향이 있다 (정부기관이나 국제관계의 특정 거래에서는 두 거래 당사자간에 주고 받는 메세지에 대한 송신자 인증만이 요구되며 비밀 채널이 형성되는 것을 용납할 수 없는 경우가 종종 있다). 이는 미국(DSA: Digital Signature Standard[31])이나 러시아(GOST[16]) 등의 국가표준 서명방식이 ElGamal형의 방식을 채택한 이유의 하나가 아닐까 생각된다. ElGamal형의 서명방식은 사전계산을 이용하면 서명의 생성에는 거의 시간이 걸리지 않는다는 장점이 있다. 반면 검증에는 RSA에 비해 상당히 많은 시간이 소요된다.

디지탈서명의 검증을 위해 검증자는 서명자의 올바른 공개 검증키를 얻어야 하는데, 일반적으로 모든 사람이 인정하는 제 3자(통상 CA(Certification Authority)라 불림)가 각 개인의 ID와 그의 공개 검증키 등을 결합하여 서명/배포함으로써 공개 검증키의 소유자를 보증한다. 이와같이 공개 검증키에 대한 확인서를 통

*퓨처시스템 정보통신연구소
†포항공대 전자전기공학과
‡한국전자통신연구원
§한국정보보호센터

해 사용자의 공개 검증키와 그 사용자를 결합시킴으로써 해당 비공개 서명키에 의해 생성된 디지탈서명을 그 사용자의 것으로 연계시켜 주는 디지탈서명 방식이 확인서 이용 디지탈서명 방식이다.

본 논문에서는 그동안 국내의 암호학자들과 유관기관에 의해 추진되어 표준안으로 상정된 확인서 이용 부가형 디지탈서명의 생성과 검증을 위한 알고리듬 KCDSA (Korean Certificate-based Digital Signature Algorithm)를 기술한다. 그리고 이 KCDSA의 안전성 및 효율성, 그리고 안전한 시스템 변수의 생성 방법 등을 다루고 실제로 구현한 결과들도 제시한다.

본 논문에서 사용되는 주요 표기들은 다음과 같다:

- $A \oplus B$: 두 수 A와 B의 비트 단위의 XOR.
- $|A|$: 정수 A의 비트 길이.
- $A \parallel B$: 두 비트열 A와 B의 연접(concatenation).
- $\mathbf{Z}_N = \{0, 1, \cdots, N-1\}$.
- $\mathbf{Z}_N^*$: 0과 N 사이의 수 중 N과 서로소인 수들의 집합.
- $K \in_r S$: 집합 S에서 K를 랜덤하게 선택함.

제 2 절 시스템 및 사용자 초기화

2.1 시스템 초기화

서명 시스템의 공개변수로 소수 P, Q, 기본원소 G 그리고 해쉬함수 h가 먼저 결정되어야 한다. P, Q, G는 적용 대상 영역 안의 모든 사용자들이 공용으로 사용할 수도 있는 시스템 변수이고, 그 크기는 적용 목적에 따라 다르게 정할 수 있다 (4절 참조). P, Q, G 및 h는 다음의 조건을 만족하도록 선정한다.

- 소수 P : 512비트부터 시작, 256비트 단위씩 증가하여 2048비트까지의 비트길이를 가지는 소수 (즉 $|P| = 512 + 256i$, $0 \le i \le 6$). 안전성을 위하여 $(P-1)/2Q$이 역시 소수이거나 $(P-1)/2$이 최소한 Q보다 큰 소수들의 곱으로 구성되는 소수 P를 사용할 것을 권고한다 (자세한 생성 방법은 부록 참조).
- 소수 Q : $P-1$을 나누는 소수로 128비트부터 시작, 32비트 단위로 증가하여 최대 256비트까지의 크기를 가질 수 있다 (즉 $|Q| = 128 + 32j$, $0 \le j \le 4$).
- 기본원 G : 위수 (order) Q를 갖는 기본원소. 즉 $G \neq 1$이고 $G^Q = 1 \bmod P$인 G. 임의의 수 a를 선택하여 $a^{(P-1)/2Q} \neq 1 \bmod P$이면 $G = a^{(P-1)/2Q} \bmod P$는 위수 Q를 갖는다.
- 해쉬함수 h : $|Q|$ 비트 길이의 출력값을 갖는 충돌저항성의 해쉬함수로 우리나라에서 표준화가 진행중인 해쉬함수를 사용할 것이 권고된다.

2.2 사용자 초기화 및 공개 검증키 등록

디지탈서명을 사용하고자 하는 각 사용자는 자신의 비공개 서명키 X 및 공개 검증키 Y를 생성하여 공개 검증키에 대해서는 공인된 인증기관 (즉 CA)으로 부터 공개키 확인서 C를 발부받아야 한다. 공개 검증키의 등록 과정은 다음과 같다 (사용자의 신원 인증이나 공개키 확인 방법 등은 본 서명 표준의 범위 밖이다).

1. 서명자는 자신의 비공개 서명키 X $(0 < X < Q)$를 랜덤하게 선택하여 비밀리에 간직한다.
2. 서명자는 비공개 서명키 X에 대응하는 공개 검증키 Y를 $Y = G^{X^{-1}} \bmod P$와 같이 계산한다. 여기서 X^{-1}는 $\bmod Q$로 X의 곱셈에 대한 역원을 나타낸다.

3. 서명자는 자신의 ID, 자신이 사용할 시스템 변수 P, Q, G와 자신의 공개 검증키 Y 등을 CA에 제시하고 그 ID가 서명자 본인임을 증명한다 (최초 등록의 경우 서명자 본인이 직접 CA를 방문하거나, 이미 비밀채널이 형성되어 있다면 이를 통해 증명할 수도 있다).

4. CA는 서명자의 신원이 확인되면 서명자가 공개 검증키 Y에 대응하는 비공개 서명키 X를 알고 있는지를 확인한다 (이러한 목적으로 영지식 증명이나 신분인증, 디지탈서명 방식 등이 이용될 수 있다).

5. CA는 단계 3의 서명자 정보를 이용하여 정해진 형식(예를들면 X.509)에 따라 서명자 인증 데이타 $Cert_Data$를 형성하고, 이에 대한 자신의 디지탈서명 C를 서명자의 공개키 확인서로 발부한다. 예를들면 $C = \{Cert\_Data \parallel Sign_{CA}(Z)\}$. 여기서 Z는 서명자 인증 데이타의 해쉬값, 즉 $Z = h(Cert_Data)$이며, $Sign_{CA}(Z)$는 Z에 대한 CA의 디지탈서명으로 반드시 KCDSA 서명을 사용할 필요는 없다 (이런 목적으로는 RSA와 같이 검증이 빠른 서명방식이 효율적이다).

표 1에 이상의 시스템 및 사용자 변수들을 정리하였다.

변 수	조 건 / 값	설 명
P	$\|P\| = 512 + 256i$ $(0 \le i \le 6)$	소수
Q	$\|Q\| = 128 + 32j$ $(0 \le j \le 4)$	$P-1$의 소인수
G	$G \neq 1$ & $G^Q = 1 \bmod P$	기본원소
h	출력길이 $= \|Q\|$	충돌회피성 해쉬함수
X	$0 < X < Q$	서명자의 비공개 서명키
Y	$Y = G^{X^{-1}} \bmod P$	서명자의 공개 검증키
Z	$Z = h(Cert\_Data)$	서명자 인증정보의 해쉬값
C	$C = \{Cert\_Data \parallel Sign_{CA}(Z)\}$	서명자의 공개키 확인서

표 1: 시스템 및 사용자 변수

제 3 절 서명 생성 및 검증 과정

3.1 서명 생성 과정

서명과정에서는 메시지 M을 받아 서명된 메시지 $\{M \parallel \Sigma\}$를 출력한다. 서명 Σ는 $\{R \parallel S\}$이다.

1. 난수값 K를 $\{1, \cdots, Q-1\}$에서 랜덤하게 선택한다 (간단한 표기로 $K \in_r \mathbf{Z}_Q^*$).

2. 증거값 $W = G^K \bmod P$를 계산한다.

3. 서명의 첫 부분 $R = h(W)$을 계산한다.

4. 메시지의 해쉬코드 $H = h(Z \parallel M)$을 계산한다.

5. 중간값 $E = R \oplus H \bmod Q$를 계산한다.

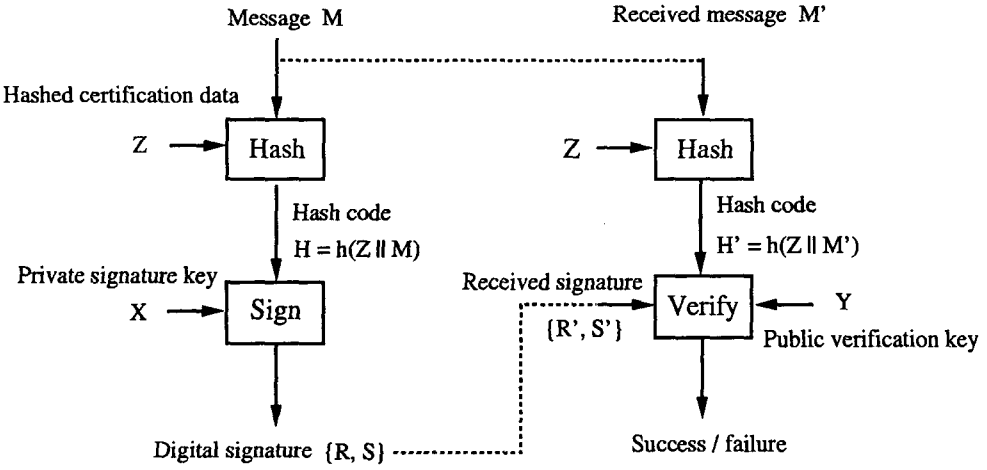
6. 서명의 두 번째 부분 $S = X(K - E) \bmod Q$를 계산한다.

7. 서명 $\Sigma = \{R \parallel S\}$를 만들어 서명된 메시지 $\{M \parallel \Sigma\}$를 출력한다.

단계 1부터 단계 3까지의 과정은 서명할 메시지와 관계가 없으므로 사전에 계산해 둘 수도 있다. 즉 K와 R을 미리 계산해서 보관하고 있다가 서명할 메시지가 들어오면 단계 4부터 실시간 계산을 할 수 있다. 이런 사전계산 방식은 스마트카드와 같이 계산력이 약한 휴대용 장치나 다수의 서명을 실시간으로 계산할 필요가

있는 응용에서 유용하게 사용될 수 있다. 이를 위해서는 사전 계산된 $\{K, R\}$을 저장할 여분의 메모리를 갖추어야 한다.

단계 3과 6에서 R 또는 S가 0인지를 검사할 수도 있다. 이 경우 만일 $R = 0$이거나 $S = 0$이면 단계 1로 되돌아 가야 한다. 그러나 R이나 S가 0이 될 확률은 R이나 S가 다른 특정 값을 가질 확률과 동일하며, 또한 R이나 S가 0이 된다고 하더라도 서명을 위조하거나 비공개 서명키 X을 얻는데 도움이 되지는 않는다.



비공개 서명키 : X, 공개 검증키 : $Y = G^{X^{-1}} \bmod P$	
서명 생성	서명 검증
$K \in_r \mathbb{Z}_Q^*$, $H = h(Z \parallel M)$	$H' = h(Z \parallel M')$
$R = h(G^K \bmod P)$	$E' = R' \oplus H' \bmod Q$
$E = R \oplus H \bmod Q$	$R' = h(Y^{S'} G^{E'} \bmod P)$?
$S = X(K - E) \bmod Q$	

그림 1: 서명 생성 및 검증 과정

3.2 서명 검증 과정

먼저 서명을 검증하기 전에 검증자는 서명자의 공개 검증키 Y와 시스템 변수 P, Q, G 등에 대한 확인된 값들을 얻어야 한다. 이들은 통상 공개키 확인서 $C = \{Cert_Data \parallel SiG_{CA}(Z)\}$의 형태로 제공된다. 이를 얻으면 검증자는 우선 $Z = h(Cert_Data)$를 계산하여 CA의 서명을 확인한 다음, $Cert_Data$로부터 서명 검증과정에서 사용할 시스템 변수 P, Q, G와 공개 검증키 Y를 추출한다.

1. 서명된 메시지 $\{M' \parallel \Sigma'\}$로 부터 검증할 메시지 M', 서명의 첫 부분 R', 서명의 두 번째 부분 S'를 추출한다. 이 때 $0 < R' < 2^{|Q|}$이고 $0 < S' < Q$임을 확인한다.
2. 검증할 메시지의 해쉬코드 값 $H' = h(Z \parallel M')$을 계산한다.
3. 중간값 $E' = R' \oplus H' \bmod Q$을 계산한다.
4. 서명자의 공개 검증키 Y를 이용하여 증거값 $W' = Y^{S'} G^{E'} \bmod P$를 계산한다.
5. $h(W') = R'$이 성립하는지 확인한다.

단계 1과 5의 확인과정이 모두 통과되면 수신된 서명 Σ'는 공개 검증키 Y에 대응하는 비공개 서명키 X를 소유한 서명자의 메세지 M'에 대한 서명임이 판명된 것이다. 단계 1과 5에서 하나라도 그 검증이 실패하면 메시지 M'에 불법적인 방법으로 서명이 되었거나 메시지가 변경된 것이므로 다음 단계로 넘어갈 필요도 없이 M'에 대한 서명이 거짓임이 밝혀진 것이다.

올바른 서명과 메세지가 수신되었을 때, 즉 $M' = M$, $R' = R$, $S' = S$일 때, $h(W') = R'$이 성립하는 것은 다음과 같이 보일 수 있다: $H' = h(Z \parallel M') = h(Z \parallel M) = H$이고, $E' = R' \oplus H' \bmod Q = R \oplus H \bmod Q = E$이다. 따라서 $S' = S = X(K - E) \bmod Q$이므로

$$\begin{aligned} W &= Y^{S'} G^{E'} \bmod P \\ &= Y^{S} G^{E} \bmod P \\ &= Y^{X(K-E) \bmod Q} G^{E} \bmod P \\ &= G^{X^{-1} \cdot X(K-E) \bmod Q} G^{E} \bmod P \\ &= G^{(K-E) \bmod Q} G^{E} \bmod P \\ &= G^{K} \bmod P \end{aligned}$$

따라서, $h(W') = h(G^K \bmod P) = R$이고, $R' = R$이므로 $h(W') = R'$이다.

그림 1에 이상에서 기술된 서명의 생성 및 검증 과정을 간략히 도시하였다.

제 4 절 안전성 분석

4.1 P, Q의 길이에 따른 안전성

KCDSA는 유한체 GF(P)위에서 정의된 알고리듬이며 KCDSA의 안전도는 유한체에서 이산대수 문제를 풀기 어렵다는 사실에 기초하고 있다. 이산대수 문제란 GF(P)의 원소 G ($G^Q \bmod P = 1$, $G \neq 1$을 만족)로 생성되는 부분 곱셈군 $< G >$의 주어진 원소 Y에 대하여 $Y = G^X \bmod P$를 만족하는 X를 찾는 문제이며, 이러한 이산대수 문제가 쉽게 풀리면 본 디지탈서명 알고리듬은 전혀 안전하지 않게 된다. 따라서 이산대수 문제를 푸는 것이 계산상 불가능하도록 소수 P와 Q의 크기를 선택해야 한다.

지금까지 알려진 이산대수문제를 푸는 범용 알고리듬 중 가장 빠른 알고리듬은 GNFS (Generalized Number Field Sieve)이다 [10]. 한편 Q가 작은 특수한 경우에는 Pollard의 ρ 알고리듬[26]을 사용하는 것이 보다 효과적일 수 있다. GNFS는 그 계산 복잡도가 소수 P의 크기에만 의존하는 범용 알고리듬이다. Pollard의 ρ 알고리듬의 계산 복잡도는 주로 Q의 크기에 의존하나 소수 P의 크기도 약간의 영향을 미친다. 그러나 그 정도는 그렇게 크지 않으므로 보통 Q의 크기만으로 그 계산 복잡도를 추정한다.

1994년을 기준으로 전문가들이 예측한 GNFS 및 Pollard의 ρ 알고리듬의 계산 복잡도는 표 2과 같다[20]. 여기서 MY는 Mips-Years의 약자이고 1 Mips-Year는 대략 1 MIPS (Million Instructions Per Second)의 프로세서가 1년간 행할 수 있는 계산량을 의미한다. 이러한 예측은 현재의 알고리듬과 기술발달 정도를 기준으로 산정된 것이며, 향후 획기적인 알고리듬이나 하드웨어의 발전이 없는 한 어느 정도는 이산대수문제의 난이도에 따른 파라미터의 크기 선정에 참고가 될 것으로 생각된다.

$\lvert P\rvert$	512	768	1024	1280	1536	2048
MY	$3 \cdot 10^4$	$2 \cdot 10^8$	$3 \cdot 10^{11}$	$1 \cdot 10^{14}$	$3 \cdot 10^{16}$	$3 \cdot 10^{20}$
* 1994년 기준 2014년의 예상되는 가용 계산력 : $10^{11} \sim 10^{13}$ MY						
* $\lvert Q\rvert = 160(\lvert P\rvert = 1024)$에 대한 Pollard의 ρ 법의 계산 복잡도 : $> 10^{14}$ MY						

표 2: 안전도에 따른 $|P|, |Q|$의 선택 예

이러한 예측을 근거로 현재 디지탈서명에 사용되는 소수 P, Q의 크기에 대한 권고치는 대략 $|P| = 1024$, $|Q| = 160$ 정도이다. 그러나 매우 중요한 응용에 사용되는 서명의 경우는 (예를들면 Root CA의 서명키) 2048비트의 P와 256비트의 Q를 사용할 수도 있겠다.

$|H|$비트의 해쉬코드를 출력하는 해쉬함수 h는 birthday paradox를 이용하면 약 $2^{|H|/2}$ 정도의 연산으로 충돌쌍을 찾을 수 있다[4]. 충돌쌍의 생성은 곧 서명의 위조를 의미하므로 이산대수 문제를 푸는 것 이외의 다른 중요한 안전 위협 요소가 된다. 따라서 해쉬함수가 birthday attack에 견디도록 출력길이 $|H|$가 선택되어야 한다. Birthday attack에 요구되는 연산 수 $2^{|H|/2}$는 대략 $|H|$비트 길이의 Q에 대한 Pollard의 ρ 알고리듬에 의한 이산대수 계산에 요구되는 연산 수와 비슷하고, 해쉬코드의 길이가 $|Q|$보다 크다고 해도 서명 생성과정에서 mod Q 연산이 사용되므로 더 큰 해쉬코드가 더 나은 안전도를 주지는 못하므로, 보통 해쉬코드의 길이를 Q의 비트길이와 같게 정한다.

4.2 서명 알고리듬의 안전성

ElGamal형의 서명 알고리듬의 안전성은 이산대수문제의 어려움에 의존하나, 과연 서명을 위조하는 것이 이산대수문제를 푸는 것 만큼 어려운가에 대한 증명은 오랫동안 암호학자들의 숙제가 되어왔다. 그러나 최근에 해쉬함수를 이상적인 랜덤 함수로 대체한다면 이러한 안전성 증명이 가능하다는 연구결과가 발표되었다 (자세한 내용은 [24, 25] 참조). 이러한 안전성 증명이 가능한 ElGamal형의 서명방식은 크게 다음의 두 부류로 나뉜다. 첫째가 Schnoor의 서명방식[29]과 같이 서명의 첫 부분을 $R = h(G^K \bmod P \parallel M)$과 같이 메세지를 증거값 $W = G^K \bmod P$와 함께 해쉬하는 것이다 (randomized hashing). 두 번째 부류는 메세지를 W와 따로 해쉬하는 경우 서명의 첫 부분을 해쉬함수를 이용하여 $R = h(W)$와 같이 계산하는 것이다. KCDSA가 후자의 부류에 속한다. 이러한 안전성 증명은 해쉬함수가 랜덤 오라클(random oracle)로 대체되었을 때에 가능한 증명이나 안전한 해쉬함수에 대해서도 어느정도의 안전성을 보장한다고 볼 수 있다.[1]

4.3 시스템 및 사용자 변수에 대한 안전성

KCDSA는 확인서를 이용한 디지탈서명 방식이다. 디지탈서명의 안전성을 위하여 검증과정에서 사용되는 공개키 및 시스템 변수들이 누구의 것인지를 확인할 필요가 있다. 이것은 사용자들 상호간에 신뢰할 수 있는 공개키 인증기관(CA)이 시스템 변수, 사용자의 공개 검증키, 사용자의 식별자, 유효기간 등이 포함된 신용장에 디지탈서명을 하여 발행해 주는 공개키 확인서를 사용함으로써 가능하다. 공개키 확인서의 발급시 CA는 반드시 사용자의 신원을 확인한 후 인증받고자 하는 공개 검증키에 대응하는 비공개 서명키를 사용자가 알고 있는지를 확인하는 과정이 포함되어야 한다. 그렇지 않을 경우 디지탈서명의 응용시 필요한 보안기능을 제공하지 못할 수도 있기 때문이다 (예를들어 타인의 공개키의 변형을 자신의 공개키로 둥록시켜 유효한 공개키 확인서를 발부받은 후 키분배나 인증 프로토콜 등에서 타인으로 가장할 수 있는 경우가 있음. 자세한 것은 [17, 13] 참조).

시스템 변수 P, Q, G는 특정 그룹내에서 합의가 되면 공통으로 사용할 수 있다. 이 경우 해당 그룹의 보안담당자는 응용분야의 중요성과 소수 P, Q의 길이에 따른 안전도 등을 고려하여 P, Q를 선택해야 하며, 소수 P, Q가 적법한 절차에 의해 생성되었다는 것을 사용자들이 검증할 수 있도록 필요한 검증 데이타를 제공해야 한다. 이는 비록 가능성이 적기는 하지만 SNFS(Special Number Field Sieve) 알고리듬을 이용하여 소수 P에 trapdoor를 설치할 수 있는 가능성을 차단하기 위함이다 ([6] 참조).

지금까지 많은 학자들이 원래의 ElGamal 방식에서와 같이 지수를 mod $P-1$로 연산하는 경우의 문제점들을 지적하고 Schnorr[29]가 제시한 소수를 위수로 갖는 원소를 기본원소로 사용할 것을 권고해 왔다[23, 22, 3, 2]. 그러나 소수를 위수로 갖는 순환군상의 이산대수문제를 이용하는 경우도 법 P를 단지 $P-1$이 큰 소수 Q만을 갖도록 랜덤하게 선택하는 경우 서명이나 키분배 등의 응용에서 안전성에 심각한 문제가 생길 수 있다 (상세한 것은 [13] 참조). 따라서 소수 P의 선택시 $(P-1)/2Q$의 소인수들의 크기가 최소한 Q보다 크게 하는 것이 바람직하다 (특히 키 분배용의 시스템 변수라면 반드시 이 조건을 지켜야만 안전하게 된다). 가장 좋은 것은 소수 P가 $P = 2QR + 1$ (R도 소수)의 형태를 갖는 것이다. 이러한 소수 P, Q를 생성하는 한 방법을 부록에 제시하였다.

[1] 안전한 해쉬함수를 이용하여 가장 근접한 랜덤함수를 만드는 방법은 해쉬함수의 입력의 일부로 비밀키를 사용하는 것이다 (keyed hash function). 그러나 해쉬함수는 서명을 검증하고자 하는 모든 사람들이 계산할 수 있어야 하므로 현실적으로 가능한 한 방법은 안전한 물리적 장치(tamper proof device)를 이용하는 것이다. [25] 참조.

파라미터 Z는 P,Q,G 등의 서명자가 사용하는 시스템 변수와 서명자의 ID와 공개 검증키 Y등 서명자과 관련된 모든 변수들에 대한 정보를 함축하고 있다. 이를 메세지와 함께 해쉬한 결과(즉 $H=h(Z\parallel M)$)를 서명함으로써 위조된 공개키의 등록이나 시스템 변수 Q 생성시의 조작[30] 등에 대한 위협을 방지할 수 있다. 이러한 위협은 디지탈 서명방식 자체의 안전성과는 별개로 파라미터 선정 및 등록 과정에서의 부주의에 기인하는 것으로 실제 서명방식의 구현시 안전성을 위해 매우 중요한 고려사항이 된다.

해쉬코드의 생성시 Z와 메세지 M의 순서도 중요한 의미를 갖는다. $H=h(Z\parallel M)$과 같이 계산하는 것과 $H=h(M\parallel Z)$와 같이 계산하는 것은 대부분의 경우 별 차이를 주지 않는다. 그러나 만일 해쉬함수 h에 대해 월등한 자본력과 계산력을 가진 집단이 특정 메세지에 대한 충돌쌍을 찾으려고 하는 경우를 가정해 보자 (해쉬함수에 대한 약간의 결함이 밝혀지거나 위조하고자 하는 메세지가 충분한 투자가치가 있다는 판단하에 특수 하드웨어 및 병렬처리 기법 사용; [21] 참조). 이 경우 $H=h(M\parallel Z)$와 같이 해쉬코드를 생성한다면 일단 원하는 메세지에 대한 충돌쌍을 찾으면 이 충돌쌍으로 모든 서명자에 대한 서명위조가 가능하다. 반면 $H=h(Z\parallel M)$과 같이 계산하는 경우는 Z값이 각 서명자마다 다르므로 충돌쌍 역시 특정 개인을 상대로 찾아야 한다. 따라서 전자의 경우와 같은 전면적인 서명 위조는 불가능하게 된다.

디지탈 서명용의 시스템 변수 P,Q,G는 이산대수문제에 바탕을 둔 Diffie-Hellman 형의 키분배 방식을 위해서도 사용 가능하다. 그러나 같은 시스템 변수를 서로 다른 용도에 공용으로 사용하는 것은 피하는 것이 바람직하다. 한 시스템의 안전도가 다른 시스템의 안전도에 영향을 주지 않도록 설계하는 것이 바람직하다는 것을 상기하면 이는 당연한 사실이다. 또한 디지탈서명은 특성상 다른 암호시스템에 비해 그 중요도가 더 높고 유효기간이 길 수 있으므로 키분배나 인증 등에서보다 높은 수준의 안전도를 갖는 시스템 변수를 사용하는 것이 일반적이다.

제 5 절 기존 방식들과의 비교

이산대수문제에 바탕을 둔 ElGamal형의 디지탈서명 방식은 그동안 많은 사람들에 의해 연구되어 왔고, 이들은 모두 다음과 같은 일반화된 서명식으로부터 얻어질 수 있다 (참고문헌 [33, 19, 7, 8] 참조). 서명의 첫 부분인 R은 함수 f에 의해 다음과 같이 계산된다:

$$R=f(G^K \bmod P,\ h(M))\in \mathbf{Z}_Q.$$

여기서 함수 f의 두 번째 인수인 $h(M)$은 생략되어 첫 인수만을 가질수도 있고, 이 경우는 f가 첫 인수값 그 자체가 될 수도 있다 (identity function). 서명의 두 번째 부분인 S는 다음과 같은 일반적인 서명 방정식을 풀어서 얻을 수 있다:

$$AK+BX+C=0 \bmod Q.$$

여기서 계수 A,B,C는 $R,S,h(M),1$ 혹은 이들의 함수가 될 수 있고, S는 위의 서명식을 S에 대해 풀어서 구한다. 표 3에 대표적인 ElGamal형 서명방식들을 정리하였다.

KCDSA에서 서명키와 검증키의 형태가 다른 방식과 달리 표기되었으나 X를 $X^{-1} \bmod Q$로 대치하면, 즉 검증키를 $Y=G^X \bmod P$로 두면, 서명식과 검증식은 다음과 같이 되어 다른 방식과 유사한 형태가 된다.

$$\begin{aligned} S &= X^{-1}(K-R\oplus H) \bmod Q, \\ R &= h(Y^S G^{R\oplus H} \bmod Q). \end{aligned}$$

세개의 국가표준 서명방식인 DSS, GOST, KCDSA를 서로 비교해 보자. 우선 소수 P,Q의 길이를 살펴보자 (표 4참조). Q의 길이는 곧 서명에 사용되는 해쉬함수의 출력길이를 의미한다. 소수 P의 길이는 DSS에서는 512비트부터 1024비트까지 64비트 단위로 증가시킬 수 있게 되어 있고, GOST에서는 512 혹은 1024비트 두 개로 고정되어 있다. KCDSA에서는 512비트에서 2048비트까지 256비트 단위로 증가시킬 수 있게 하였다. 현재 대부분의 응용에서 권장되고 있는 P의 길이는 안전도에 따라 768, 1024 및 2048비트 정도임을 감안하면 256비트 단위의 증가면 충분하다고 생각된다. Q의 길이는 DSS와 GOST에서는 160 및 256비트로 고정되어 있으나 KCDSA에서는 128비트부터 32비트 단위로 증가하여 최대 256비트까지 가능하게 하였다. 이는 응용에 따라 적절하다고 생각되는 Q의 크기를 선택 가능하게 하기 위함이나 현재로서는 대부분의 응용에

서명 방식	서명 생성	서명 검증
서명키 : $X \in_r \mathbf{Z}^*_{P-1}$		검증키 : $Y = G^X \bmod P$
ElGamal [5]	$K \in_r \mathbf{Z}^*_{P-1}$, $H = h(M)$ $R = G^K \bmod P$ $S = K^{-1}(H - RX) \bmod P-1$	$Y^R R^S = G^H \bmod P$?
AMV [1]	$K \in_r \mathbf{Z}_{P-1}$, $H = h(M)$ $R = G^K \bmod P$ $S = X^{-1}(H - RK) \bmod P-1$	$Y^S R^R = G^H \bmod P$?
서명키 : $X \in_r \mathbf{Z}_Q$		검증키 : $Y = G^X \bmod P$
Schnorr [29]	$K \in_r \mathbf{Z}_Q$ $R = h(G^K \bmod P \parallel M)$ $S = K - RX \bmod Q$	$h(G^S Y^R \bmod P \parallel M) = R$?
Nyberg-Rueppel [19]	$K \in_r \mathbf{Z}_Q$, $H = h(M)$ $R = (G^K \bmod P) \bmod Q$ $S = K - RXH \bmod Q$	$(G^S Y^{RH} \bmod P) \bmod Q = R$?
DSS [31]	$K \in_r \mathbf{Z}^*_Q$, $H = h(M)$ $R = (G^K \bmod P) \bmod Q$ $S = K^{-1}(RX + H) \bmod Q$	$(Y^{S^{-1}R} G^{S^{-1}H} \bmod P) \bmod Q = R$?
GOST [16]	$K \in_r \mathbf{Z}_Q$, $H = h(M)$ $R = (G^K \bmod P) \bmod Q$ $S = RX + KH \bmod Q$	$(Y^{-RH^{-1}} G^{SH^{-1}} \bmod P) \bmod Q = R$?
서명키 : $X \in_r \mathbf{Z}^*_Q$		검증키 : $Y = G^{X^{-1}} \bmod P$
KCDSA	$K \in_r \mathbf{Z}^*_Q$, $H = h(Z \parallel M)$ $R = h(G^K \bmod P)$ $S = X(K - R \oplus H) \bmod Q$	$h(Y^S G^{R \oplus H} \bmod P) = R$?

표 3: 대표적인 ElGamal형 디지탈서명 방식들의 비교

서 160비트 정도면 충분할 것으로 보인다. 사람 혹은 응용에 따라 선택 가능한 Q의 크기가 다양하므로 표준 해쉬함수도 이러한 Q의 크기와 같은 길이의 출력을 낼 수 있는 가변길이 출력 해쉬함수가 선정될 것이다.

서명방식	$\|P\|$	$\|Q\|$
DSS	$\|P\| = 512 + 64i$ $(i = 0, 1, \cdots, 8)$	$\|Q\| = 160$
GOST	$\|P\| = 512$ or 1024	$\|Q\| = 256$
KCDSA	$\|P\| = 512 + 256i$ $(i = 0, 1, \cdots, 6)$	$\|Q\| = 128 + 32i$ $(i = 0, \cdots, 4)$

표 4: 국가표준 서명방식들의 시스템 변수의 크기

다음으로 세 국가 표준의 서명 생성 및 검증 과정을 살펴보자. DSS의 경우는 서명 생성 및 검증 과정 모두에 mod Q로 곱셈에 대한 역원의 계산이 요구며, GOST의 경우 검증 과정에서만 역원의 계산이 필요하다. 반면 KCDSA는 설계 당시 가능한 역원의 계산을 피하도록 하는 것이 하나의 요구조건이었고, 그 결과 현재의 표준안에서는 어느 과정에서도 역원의 계산이 필요치 않다. 역원의 계산은 일반 컴퓨터에서는 확장된 유클리드 알고리듬(extended Euclid algorithm)[9, page 325]을 이용하면 모듈라 멱승에 비해 거의 무시할 수 있는 정도의 시간에 계산 가능하다. 그러나 스마트카드와 같은 제한된 자원을 갖는 환경에서는 $K^{-1} \bmod Q = K^{Q-1} \bmod Q$와 같이 역원을 흔히 모듈라 멱승을 이용하여 구현하므로 가능한 역원의 사용은 피하는 것이 바람직하다. 이러한 주장은 DSS가 표준으로 확정되기 전의 검토기간에 많은 학자들이 비판을 했던 부분 중

의 하나이다[27].

다음은 안전성의 측면을 생각해 보자. 어느 알고리듬이 안전하지 않다거나 혹은 다른 알고리듬에 비해 덜 안전하다는 등의 얘기는 하기가 어려우나, KCDSA의 경우는 적절한 가정하에 안전성을 증명하는 것이 가능하다는 잇점을 갖는다 ([24, 25] 참조). 이는 KCDSA가 암호학계의 최신 연구결과를 수용하여 설계되었기 때문이다. 또한 KCDSA는 알고리듬 자체의 안전성 뿐만 아니라 사용/응용상의 안전성을 고려하여 P, Q의 선택이나 해쉬코드의 생성 방법 등을 보강하였다.

제 6 절 구현 결과

이 논문에서 기술된 디지탈서명 표준안을 실제로 PC에서 구현하여 보았다 (Pentium 90MHz와 Pentium Pro 200MHz에서 Visual C/C++ 컴파일러 사용 ; 표 5참조).[2] 일반적으로 널리 쓰일 것으로 예상되는 768비트와 1024비트 크기의 P와 160비트 크기의 Q를 택하였고, 메세지는 단문을 택해 서명 생성 및 검증에 걸리는 시간이 실제로 모듈라 연산에 드는 시간을 반영하도록 하였다 (해쉬함수는 SHA-1[32]을 사용하였음).

서명의 생성에 걸리는 시간은 거의 $G^K \bmod P$의 계산에 걸리는 시간과 같고, 이는 잘 알려진 윈도우 멱승 알고리듬을 이용할 경우 160비트 크기의 지수에 대해 평균 196번의 모듈라 곱셈이 요구된다 (윈도우 크기 = 4). 만일 128개의 $|P|$비트 크기의 사전계산값들을 이용한다면 필요한 모듈라 곱셈 수는 약 37개로 줄일 수 있다 (물론 더 큰 테이블을 이용하는 경우 곱셈수를 훨씬 줄일 수 있다. 예를들어 1024개의 G에 대한 사전계산값들을 이용하는 경우 약 23번의 모듈라 곱셈이면 $G^K \bmod P$의 계산이 가능하다. 자세한 것은 [12] 참조.) 이는 1024비트 크기의 P에 대해서 단지 16Kbytes 정도의 메모리만을 이용한 것이나, 효율성은 거의 5배 정도로 높일 수 있음을 의미한다. 또한 안전한 저장소가 있다면 K와 $R = h(G^K \bmod P)$를 사전에 구하여 저장해 둠으로써 서명의 생성에는 거의 시간이 걸리지 않게 할 수도 있다.

서명의 검증에 걸리는 시간은 $Y^S G^{R \oplus H} \bmod P$에 걸리는 시간과 거의 같다. 이런 형태의 모듈라 멱승 역시 윈도 멱승법을 사용하면 160비트 크기의 지수에 대해 평균 238번의 모듈라 곱셈이면 계산 가능하다 (윈도우 크기 = 2). 이는 서명 생성에 필요한 196번에 비해 단지 20% 정도 증가한 계산량이다.

기종		Pentium/90		Pentium Pro/200	
$\|P\|$	Lang.	서명생성	서명검증	서명생성	서명검증
768	C	174	220	56	71
	D	93	111	30	37
	A	42	51	11	13
1024	C	295	370	95	115
	D	160	198	51	62
	A	60	77	17	20

Language :
C = C only,
D = C with double digit option(_int64),
A = C with partial assembly.

표 5: 서명 표준안의 구현 결과 ($|Q| = 160$; 단위: msec)

표에서 볼 수 있듯이 1024비트 크기의 P에 대해 C언어만으로도 Pentium 90MHz에서 0.2초 이하로 서명 생성이나 검증이 가능하고, 부분적으로 어셈블리어를 사용한다면 이보다 2배 이상 속도를 향상시킬 수 있음

[2]서명의 구현에는 (주)퓨처시스템 정보통신연구소 보안팀에서 개발한 Crypto Library를 사용하였으며, 표의 수치는 Montgomery 모듈라 감소 알고리듬[18]과 윈도우 멱승 알고리듬을 사용한 결과임. 윈도우 멱승 알고리듬 및 그 효율성 분석에 대해서는 참고문헌 [11, Section 1.2]을 참조하고, 모듈라 연산의 보다 자세한 구현 결과는 [14, 15]을 참조하기 바람.

을 알 수 있다.[3] 서명 생성의 경우는 만일 $G^K \bmod P$의 계산에 사전계산 테이블을 이용할 경우 메모리 요구량에 따라 약 3 ~ 6배 정도 더 빠른 속도를 낼 수 있다. 따라서 대부분의 응용에서 서명의 생성이나 검증에 걸리는 시간은 거의 문제가 되지 않을 것이며, 오히려 긴 메세지의 경우 대부분의 시간은 메세지를 해쉬하는데 소요될 것이다.

제 7 절 결 론

본 논문에서는 현재 우리나라의 디지탈서명 표준안으로 상정된 서명 알고리듬을 기술하고, 안전성 분석 및 파라미터 선정 방법, 구현결과 등을 제시하였다. 향후 이 표준안은 일정기간의 검토기간을 거쳐 표준으로 확정될 것이며, 검토기간 동안 부분적으로 표준안에 약간의 변경은 있을 수 있으나 서명 알고리듬 자체의 변경은 없을 것으로 생각된다. 이제 이 표준안이 국가 표준으로 확정되면 인터넷 보안이나 전자상거래, EDI 등 많은 응용분야에서 매우 유용하게 사용될 것으로 믿는다.

감사의 말: KCDSA는 지난 3년여의 기간 동안 국내 암호학계의 많은 분들이 참여하여 토론하는 과정을 거쳐 현재의 표준안으로 정착된 것이며, 특히 이 은정(포항공대), 문 상재(경북대), 원 동호(성균관대), 장 청룡(경동대), 이 경석(산업연구원), 백 재현(국방과학연구소) 등 이 과정에 적극적으로 참여하신 분들께 감사드립니다.

참고 서적

[1] G.B.Agnew, R.C.Mullin and S.A.Vanstone, Improved digital signature scheme based on discrete exponentiation, *Electronics Letters*, 26(14), 1990, pp.1024-1025

[2] R.Anderson and S.Vaudenay, Minding your p's and q's, In *Advances in Cryptology - ASIACRYPT'96*, LNCS 1163, Springer-Verlag, 1996, pp.15-25.

[3] D.Bleichenbacher, Generating ElGamal signatures without knowing the secret, In *Advances in Cryptology - EUROCRYPT'96*, LNCS 1070, Springer-Verlag, 1996, pp.10-18.

[4] D.W.Davies and W.L.Price, *Security for computer networks*, 2nd edition, John Wiley & Sons, 1984, pp.262-264.

[5] T.ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. Inform. Theory*, IT-31, 1985, pp.469-472.

[6] D.M.Gordon, Designing and detecting trapdoors for discrete log cryptosystems, In *Advances in Cryptology - CRYPTO'92*, LNCS 740, Springer-Verlag, 1993, pp.66-75.

[7] P.Horster, H.Petersen and M.Michels, Meta-ElGamal signature schemes, In *Proc. 2nd ACM Conference on Computer and Communications Security*, ACM Press, 1994, pp.96-107.

[8] L.Harn and Y.Xu, Design of generalized ElGamal type digital signature schemes based on discrete logarithm, *Elect. Lett.*, 30(24), 1994, pp.2025-2026.

[9] Knuth, *The Art of Computer Programming*, Vol. 2, Addison-Wesley, 1981.

[10] A.K.Lenstra and H.W.Lenstra, *The Development of the number field seive*, Springer-Verlag, 1993.

[3] 표에서 D로 표시한 열의 수치는 Visual C/C++ 컴파일러에서 제공하는 64비트 변수 _int64를 사용했을 때의 결과이다. 부분적인 64비트 연산의 사용으로 (주로 32비트 수들의 멋셈시에 사용) 32비트 변수만을 사용했을 때에 비해 약 2배 정도 더 빠른 코드를 생성시킬 수 있음을 알 수 있다.

[11] C.H.Lim, *Computational methods for speeding up public key cryptosystems*, Ph.D Thesis, Dept. Electronic and Electrical Engineering, POSTECH, Mar. 1996.

[12] C.H.Lim and P.J.Lee, More flexible exponentiation with precomputation, In *Advances in Cryptology - Crypto'94*, LNCS 839, Springer-Verlag, pp.95-107.

[13] C.H.Lim and P.J.Lee, A key recovery attack on discrete log based schemes using a prime order subgroup, In *Advances in Cryptology - Crypto'97*, LNCS 1294, Springer-Verlag, pp.249-263.

[14] C.H.Lim, H.S.Hwang and P.J.Lee, Fast modular reduction with precomputation, *Proc. of JW-ISC'97*, pp.65-79.

[15] 황 효선, 임 채훈, 공개키 암호시스템의 고속 구현, CISC'97 자료집.

[16] M.Michels, D.Naccache and H.Petersen, GOST 34.10 - A brief overview of Russia's DSA, *Computers and Security*, 15(8), 1996, pp.725-732.

[17] A.J.Menezes, M.Qu and S.A.Vanstone, Some new key agreement protocols providing implicit authentication, In *Proc. SAC'95*, Carleton Univ., Ottawa, Ontario, May 1995, pp.22-32.

[18] P.L.MontGomery, Modular multiplication without trial division, *Math. Comp.*, 44, 1985, pp.519-521.

[19] K.Nyberg and R.Rueppel, Message recovery for signature schemes based on the discrete logarithm problem, In *Advances in Cryptology-Eurocrypt'94*, LNCS 950, Springer-Verlag, 1994, pp.183-193.

[20] A.M.Odlyzko, The future of integer factorization, *CryptoBytes*, 1(2), 1995, pp.5-12.

[21] P.C.van Oorschot and M.J.Wiener, Parallel collision search with applications to hash functions and discrete logarithms, In *Proc. 2nd ACM Conference on Computer and Communications Security*, Fairfax, Virginia, Nov. 1994, pp.210-218.

[22] P.C.van Oorschot and M.J.Wiener, On Diffie-Hellman key agreement with short exponents, In *Advances in Cryptology - EUROCRYPT'96*, LNCS 1070, Springer-Verlag, 1996, pp.332-343.

[23] S.C.Pohlig and M.E.Hellman, An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance, *IEEE Trans. Inform. Theory*, IT-24 (1), 1978, pp.106-110.

[24] D.Pointcheval and J.Stern, Security proofs for signature schemes, In *Advances in Cryptology - EUROCRYPT'96*, LNCS 1070, Springer-Verlag, 1996, pp.387-398.

[25] D.Pointcheval and S.Vaudenay, On provable security for digital signature algorithms, available from *http : //www.dmi.ens.fr/~pointche/*.

[26] J.M.Pollard, Monte Carlo methods for index computation (mod p), *Math. Comp.*, 32(143), 1978, pp.918-924.

[27] R.L.Rivest / M.E.Hellman / J.C.Anderson, Responses to NIST's proposal, *Comm. ACM*, 35(7), pp.41-52.

[28] R.L.Rivest, A.Shamir and L.Adleman, "A method for obtaining digital signatures and public key cryptosystems," *Commun. ACM*, 21(2), 1978, pp.120-126.

[29] C.P.Schnorr, Efficient signature generation by smart cards, *Journal of Cryptology*, 4(3), 1991, pp.161-174.

[30] S.Vaudenay, Hidden collisions on DSS, In *Advances in Cryptology - CRYPTO'96*, LNCS 1109, Springer-Verlag, 1996, pp.83-88.

[31] NIST, Digital signature standard, *FIPS PUB 186*, 1994.

[32] NIST, Secure hash standard, *FIPS PUB 180-1*, Department of Commerce, Washington D.C., Apr. 1995.

[33] ISO/IEC CD 14888-3, Digital signatures with appendix - Part 3: Certificate-based mechanisms.

부록 A 안전한 소수 P,Q의 생성

KCDSA는 다음의 조건을 만족하는 두 소수 P,Q을 필요로 한다.

a. $2^{|P|-1} < P < 2^{|P|}$, $|P| = 512 + 256i$ $(0 \le i \le 6)$.

b. $2^{|Q|-1} < Q < 2^{|Q|}$, $|Q| = 128 + 32j$ $(0 \le j \le 4)$.

c. Q는 $P-1$을 나누고, $(P-1)/2Q$는 소수이거나 Q보다 큰 소인수들의 곱.

조건 a, b는 소수 P,Q가 가질 수 있는 크기를 규정한 것이며, 조건 c는 $P-1$이 Q 보다 작은 소인수들을 가질 때 가능한 공격들을 막기 위한 조건이다[13]. 여기서는 $R = (P-1)/2Q$도 역시 소수가 되도록 P,Q를 생성하는 방법을 기술한다. 즉 $P = 2QR+1$을 만족하는 소수 P,Q,R를 찾으면 된다. 이때 R이 Q보다 훨씬 큰 소수이므로 효율성을 위하여 먼저 소수 R를 선택한 후, 여러개의 소수 Q에 대해서 $2QR+1$이 소수인지를 판정하는 방법을 취한다.

이 절에서는 위의 조건을 만족하는 소수 P,Q와 기본원소 G를 생성하는 일반적인 알고리듬을 기술한다. 소수 P와 Q의 생성시 필요한 난수의 발생에는 $|Q|$비트의 출력을 내는 해쉬함수 h를 사용할 수 있다. 설명의 편의상 주어진 seed s로 n비트의 출력을 내는 의사난수발생기를 다음과 같이 정의하자. 여기서 $k = \frac{n}{|Q|}$, $r = n \bmod |Q|$를 나타낸다.

$$\begin{aligned} PRG(s,n) &= V_k \| V_{k-1} \| \cdots V_1 \| V_0, \text{ where} \\ V_i &= h(s+i) \text{ for } i = 0, 1, \cdots, k, \\ V_k &= h(s+k) \bmod 2^r. \end{aligned}$$

A.1 소수 P,Q 및 기본원소 G 생성 알고리듬

다음은 위의 조건을 만족하는 소수 P,Q와 기본원 G를 생성하는 알고리듬이고, 여기서 사용되는 강한 소수 판정 알고리듬으로 Miller-Rabin의 확률적인 소수 판정 알고리듬[9, page 379]을 이용할 수 있다.[4]

1. 최소한 $|Q|$비트가 되도록 임의의 비트열 s을 선택한다.

2. $tCount = rCount = 1$, $pCount = qCount = gCount = 0$으로 둔다.

3. 다음과 같이 $Seed$를 형성한다:

$$\begin{aligned} Seed &= s \| w_2 \| w_1 \| w_0, \text{ where} \\ w_2 &= 0x00 \| i \| j \| tCount, \\ w_1 &= rCount \mid pCount, \\ w_0 &= qCount \| gCount \| 0x00. \end{aligned}$$

[4] 강한 소수 판정 알고리듬은 소수가 아닌 정수가 알고리듬을 통과할 확률이 2^{-80}이하이어야 한다.

여기서 i, j는 각각 P, Q의 크기 $|P| = 512 + 256i$, $|Q| = 128 + 32j$에서 i, j를 8비트씩으로 나타낸 것이며, $tCount$와 $gCount$는 8비트, $pCount$, $qCount$ 및 $rCount$들은 모두 16비트 크기를 갖는다. 이후 각 $Count$가 변경될때는 이에 따라 $Seed$도 변경되는 것으로 가정한다.

4. $Seed$를 이용하여 $|P| - |Q| - 1$비트의 난수 U를 발생시킨다. 즉

$$U = PRG(Seed,\ |P| - |Q| - 1).$$

5. U의 최상위 및 최하위 비트를 1로 만들어 이를 R로 둔다. 즉

$$R = 2^{|P|-|Q|-2} \vee U \vee 1.$$

여기서 $\vee$는 비트 단위의 논리합을 의미한다.

6. 강한 소수 판정 알고리듬으로 R을 판정하여 소수이면 단계 9로 간다.

7. $rCount$를 1 증가시킨다.

8. $rCount < 2048$이면 단계 4로, 그렇지 않으면 단계 1로 간다.

9. $pCount = 1$, $qCount = 1$로 둔다.

10. $Seed$를 이용하여 $|Q|$비트의 난수 V를 만든다.

$$U = PRG(Seed,\ |Q|).$$

11. V의 최상위 비트와 최하위 비트를 1로 만든 수를 Q로 둔다. 즉

$$Q = 2^{|Q|-1} \vee V \vee 1.$$

12. $2QR + 1$의 비트수가 $|P|$보다 작으면 단계 14로 간다.

13. 강한 소수 판정 알고리듬으로 Q를 판정하여 소수이면 단계 16으로 간다.

14. $qCount$를 1 증가시킨다 (즉 $qCount = qCount + 1$).

15. $qCount < 1024$이면 단계 10으로, 그렇지 않으면 단계 17로 간다.

16. 강한 소수 판정 알고리듬으로 $P = 2QR + 1$을 판정하여 소수이면 단계 21로 간다.

17. $pCount$를 1 증가시키고, $qCount = 1$로 둔다.

18. $pCount < 4096$이면 단계 10으로 간다.

19. $tCount$를 1 증가시킨다 (즉 $tCount = tCount + 1$).

20. $tCount < 256$이면 단계 3으로, 그렇지 않으면 단계 1로 간다.

21. $gCount = 1$로 둔다.

22. $Seed$를 이용하여 $|P|$비트 길이의 난수 U를 만든다. 즉

$$U = PRG(Seed,\ |P|).$$

23. $G = U^{(P-1)/2Q} \bmod P$를 계산한다.

24. 만일 $G \neq 1$이면 단계 26으로 간다.

25. $gCount$를 1 증가시키고 단계 22로 간다.

26. 소수 P, Q와 기본원소 G, 증거값 $Seed$를 출력한다.

위의 알고리듬에서 $Seed$의 주 성분인 비트열 s는 랜덤수일 필요는 없으며, 시스템 변수를 사용할 개인이나 단체에 대한 정보 등이 이용될 수도 있다. P, Q, G와 함께 출력되는 $Seed$는 이들이 위 알고리듬에 의해 적절히 생성되었다는 사실을 확인하는데 사용될 수 있다.

$Seed$에 포함된 각 $Count$값들은 해당 P, Q, G를 생성하는데 소요된 시간에 대한 거의 모든 정보들을 포함하고 있다. 즉 $tCount$는 위의 알고리듬 전체가 몇 번이나 돌았는 지를 나타내며 (대부분의 경우 $tCount = 1$일 것임), $rCount$는 소수 R의 생성에 소요된 $|R|$비트 크기의 수에 대한 소수판정 알고리듬의 수행 수를, $qCount$는 소수 Q의 생성에 소요된 $|Q|$비트 크기의 수에 대한 소수판정 알고리듬의 수행 수를, 그리고 $pCount$는 소수 P를 찾기 위해 생성했던 $|Q|$비트 크기의 소수의 수 및 $|P|$비트 크기의 수에 대한 소수판정 알고리듬의 수행 수를 각각 나타낸다. 마지막으로 $gCount$는 기본원소 G를 찾는데 필요한 $(P-1)/2Q$ 정도 크기의 수에 대한 모듈라 멱승의 수를 나타낸다 (소수 P를 $R = (P-1)/2Q$ 역시 소수가 되도록 했으므로 대부분의 경우 $gCount = 1$일 것임).

A.2 수치 예

다음은 위의 소수생성 알고리듬을 구현하여 얻은 값들이다 ($|P| = 768$과 $|P| = 1024$, $|Q| = 160$으로 동일). $Seed$는 총 256비트로 첫 5 워드는 π의 소수부분의 첫 160비트를 딴 것이며, 나머지 3 워드는 알고리듬에서 기술한 대로이다 (해쉬함수는 SHA-1을 사용하였음).

```
Seed = 243F6A88 85A308D3 13198A2E 03707344 A4093822 00010101 0116032C 000D0100
   P = 8745EC1A BB9DDFDC 466D7AB8 63FCBFC4 911CB370 B7AECEC7 2FA5C0F8 AF4BC55A
       55F6DE0F 06AA1DB2 B7EDBF9E 875B1EF8 D5ACD52B A72C2065 822612BB B7416529
       B263BFB0 3E31164D 5553CB73 9CF67783 49E4546D 5F1DFC9B AC68ACE8 99F6E9EB
   Q = BFEFB8A9 71882755 C3A97772 1F175180 FC194A11
   G = 259075FA 464CCCCF DC8F62D4 67F2A142 B4D9BB8D D44ACABE C49E1A65 5F1AB861
       362609A0 3B62B5F2 75E28979 17A8F2E6 32B32053 FD5771C2 A4860924 BDF24B79
       86522F93 FC23B01B 6DB618EB A3CFABCC 0458EB63 94156C08 501F89C2 D65FCC22

Seed = 243F6A88 85A308D3 13198A2E 03707344 A4093822 00020101 02870392 00710100
   P = 8C354F55 AC22F2F9 E04FE0AD 1994B6B2 15D9A570 7DC6109F A9B68C28 26B7E888
       44529C81 AA0C1713 4EFF30AA 783EBB49 65F9DF5C ED10C7BA D4DB000A A84FADDD
       C415F3FC B0DC1980 2AF829A4 B109D2BA 469A9A19 30F5F064 E35406D3 033D7928
       462E47C0 B5BCE33A FE1FADD9 BD465DCE A78B4E12 ADC7ED82 FACF33FA 779CD00B
   Q = 98339F08 9E9A8DFF 3D5A2C3A FD9B98D7 EC703C23
   G = 5A083A31 5942F0F2 7ABEA027 2664E92E 62B5AE72 49661F7D 20ED4951 4E759397
       306553AC 68370740 75009FC8 2CD5AD2D 54AAB5D8 1AA68F31 1962A151 B4FE68E2
       EAE89D97 20E81157 D17AA025 8A8A813A C6E6B8BC 4C77F655 AE62B3ED 8FF7B831
       A12117F7 DF4F9A15 398038C4 74C6D504 797F0592 C5FB6A61 1A3B90D2 2D6DC171
```