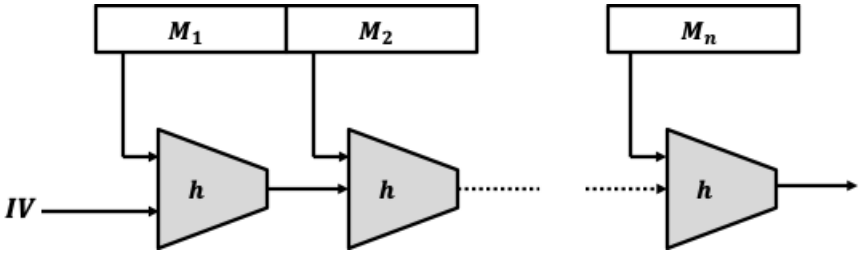
정보보호개론

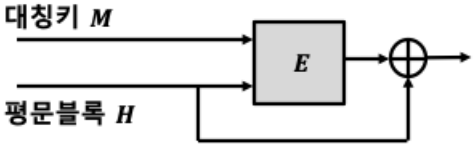
제9장 암호알고리즘: 해시함수, 공개키 암호알고리즘, 전자서명

1. 해시함수



<그림 9.1> Merkle-Damgard 충돌회피 해시함수의 구성

해시함수가 만족해야 할 가장 중요한 특성은 일방향성과 충돌회피성이다. Merkle과 Damgård는 서로 독립적으로 충돌회피 해시함수를 만드는 방법을 소개하였으며[1, 2], 오늘날 많은 해시함수가 이들이 제안한 방법에 기반하고 있다. 이들은 고정된 크기의 작은 입력을 받는 충돌회피 함수 h를 이용하여 그림 9.1과 같이 구성하면 결과 H 함수는 충돌회피 함수가 된다는 것을 증명하였다. 전체 입력은 h 함수 입력의 정확한 배수가 되도록 채우기가 이루어져야 하며, h 함수의 입력 크기로 나누어져 함수 계산이 진행된다. 해시함수 계산에서 채우기할 때 정해진 크기의 바이트에 매시지의 실제 길이를 기록한다. 이 기록은 해시함수의 정확성과 안전성에 매우 중요하다.



<그림 9.2> Davies와 Meyer의 충돌회피 함수

Merkle과 Damgård 구조에 의하면 일정 크기의 입력을 받는 충돌회피 함수 h를 만들 수 있으면 임의 크기의 메시지를 입력받는 충돌회피 해시함수를 만들 수 있다. Davies와 Meyer는 그림 9.2와 같이 블록 암호화 함수를 이용하면 충돌회피 h 함수를 만들 수 있음을 증명하였다[3]. 이 함수를 수식으로 표현하면 다음과 같다.

$$h(H, M) = E(M, H) \oplus H$$

전체 해시함수는 $IV = H_0$로 시작하여 $H_i = E.M_i(H_{i-1}) \oplus H_{i-1}$을 반복적으로 계산하는 형태로 구성하면 안전한 충돌회피 해시함수를 만들 수 있다. 전체 메시지가 블록 암호알고리즘의 키 길이로 나누어져 초깃값을 반복적으로 암호화할 때 암호키로 사용하는 형태이다. 실제 다음에 살펴볼 SHA-1과 SHA-2는 모두 이 방식으로 설계된 해시함수이다.

1.1 SHA 함수

1.1.1 SHA 역사와 특징

SHA(Secure Hash Algorithm)는 NIST에서 표준화한 해시함수이다. 표준번호 FIPS 180인 SHA 함수는 1993년에 처음 발표되었으며, Rivest가 개발한 MD5와 마찬가지로 Merkle-Damgård 구조 기반 해시함수이다. 하지만 처음 발표된 함수는 문제가 있어 이를 보완한 버전을 1995년에 발표하였으며, 이 해시함수를 표준번호 FIPS 180-1인 SHA-1이라 한다[4]. SHA-1 해시값의 길이는 160bit이다. Wang 등[5]은 2005년에 $O(2^{69})$의 비용으로 충돌을 찾을 수 있음을 보였으며, 2013년에 Stevens[6]은 $O(2^{61})$의 비용으로 충돌을 찾을 수 있음을 보였다. 해시값의 길이가 160bit이므로 충돌을 찾을 안전성은 $O(2^{80})$이 되어야 하는 것을 고려하였을 때 SHA-1은 더는 안전하게 사용하는 것이 어렵게 되었다. 실제 머지않아 충돌을 찾을 것으로 예측되고 있다. 따라서 최근에는 표준번호 FIPS 180-2인 SHA-2를 주로 사용한다. SHA-2는 해시값의 길이가 224bit, 256bit, 386bit, 512bit 버전들이 있다. 가장 많이 사용하는 것이 256bit 길이의 SHA-2이다. 현재 비트코인에서도 이 버전의 해시함수를 사용하고 있다.

SHA 계열의 해시함수에 문제점이 계속 발견됨에 따라 새 표준의 필요성이 제기되었으며, 2008년부터 새 표준에 대한 선발 작업이 AES와 유사한 방식으로 진행되었다. 2015년에 51개의 후보 중 최종적으로 Guido Bertoni 등[1]이 개발한 KECCAK[7] 이라는 알고리즘이 채택되었으며, 이를 표준번호 FIPS 202인 SHA-3이라 한다. SHA-3은 SHA-2와 동일하게 4개 길이의 해시값 출력을 지원하며, 기존 SHA 계열과는 전혀 다른 방식으로 설계되어 있다. SHA-3가 표준화되었지만 아직 SHA-2가 더 많이 활용되고 있다. 보통 새 표준을 발표하더라도 기존에 사용한 알고리즘이 심각한 허점이 있는 것이 아니면 그것을 완전 대체하는 것은 오래 시간이 보통 소요된다.

1.1.2 SHA-1의 내부

SHA-1은 2^{64}보다 작은 입력을 받아 5개의 연속된 32비트 값인 160bit 해시값을 출력한다. SHA-1은 내부적으로 512bit(64byte) 단위로 나뉘어 적용되며, 채우기를 통해 512비트의 배수를 만든다. 이때 비트 채우기(1 이후 모두 0으로 채우는 형태)를 하지만 마지막 64비트에 메시지 크기를 기록한다. 연습문제에 제시되어 있지만 이 채우기는 반드시 필요하며, 보안적으로 매우 중요한 역할을 한다.

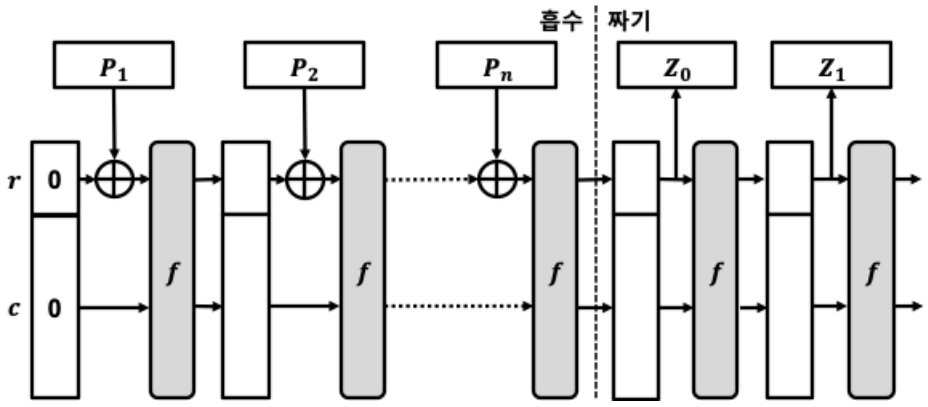
나누어진 각 512bit는 80개의 32bit 블록으로 확장되어 매 단계마다 1개가 사용된다. 초기 고정된 5개의 32bit 값을 시작으로 이 값들이 매 단계 조작되어 최종 5개의 32bit 값을 얻으며, 이것을 이용하여 다음 블록을 계산하는 방식이다. 따라서 $H(x)$값이 있을 때 x가 512bit의 배수이고 채우기를 하지 않았다면 $H(x||y)$는 $H(x)$를 초깃값으로 사용하여 $H(y)$를 계산하는 것과 같아진다.

1.1.3 SHA-3의 내부

SHA-3은 그림 9.3처럼 동작하는 스펀지 구성(sponge construction)이라는 기법을 사용한다. 이 기법은 입력 데이터를 압축한 다음 필요한 만큼의 출력을 뽑아내는 방식을 사용한다. SHA-3은 224, 256, 384, 512bit 4종류의 출력을 지원한다. SHA-3 256bit는 블록 크기가 1088bit이며, 1600bit의 입력을 받아 같은 크기 출력을 주는 내부 함수 f를 사용한다. SHA-3 256bit의 내부 동작 메커니즘은 다음과 같다.

- 단계 1. 입력 M을 채우기하여 $r(=1088)$bit의 정확한 배수가 되도록 한다. 채우기한 결과를 $p_1||\cdots||p_n$이라 하자. 여기서 $|p_i| = r$이다.
- 단계 2. 모든 비트가 0인 1600bit 길이의 S를 준비한다.
- 단계 3. 각 p_i에 대해 다음을 수행하며, 이 단계를 흡수(absorbing) 단계라 한다.

[1]KECCAK의 개발자 중에는 AES를 개발한 John Daemen도 포함되어 있다.



<그림 9.3> 스펀지 구성

- p_i가 1600bit가 되도록 0으로 채우기하고 이것을 S와 XOR한 값을 f 함수의 입력으로 사용하여 다음 S를 계산한다. 즉, $S = f(S \oplus (p_i||0...0))$이다.

- 단계 4. 빈 비트열인 Z를 준비한 다음, 다음을 수행하여 필요한 출력값을 생성한다. 이 단계를 짜기(squeezing) 단계라 한다.
  - 단계 3의 S에서 1088bit를 Z에 추가한다.
  - 필요한 출력값을 얻을 때까지 $S = f(S)$를 계산하며, 각 단계마다 1088bit를 Z에 추가한다.

SHA-3 256bit는 흡수 단계의 최종 S에서 256bit를 취하면 최종 해시값을 얻을 수 있다. SHA-3는 10*1 형태의 채우기를 사용한다. 이전 SHA와 달리 끝에 메시지 길이를 추가하지 않으며, 채우기가 최소 2비트가 필요하다.

2. MAC

MAC은 메시지에 대한 무결성 서비스를 암호학적으로 제공하기 위해 사용하는 암호알고리즘이다. MAC은 일반 해시함수와 달리 메시지와 암호키, 두 개의 입력을 사용한다. 따라서 키가 없는 사용자는 MAC 값을 생성할 수 없으며, 확인도 할 수 없다. MAC 함수는 보통 대칭 암호알고리즘이나 해시함수를 사용하여 만든다. 대칭 암호알고리즘을 이용하여 만드는 방법에는 CBC-MAC, CMAC(Cipher-based MAC)[8], NMAC(Nested MAC), PMAC(Parallelizable MAC)[9] 등의 방법이 있으며, 해시함수를 이용하는 MAC을 HMAC(Hash-based MAC)이라 한다.

참고로 해시후 해시값을 대칭암호알고리즘으로 암호화하거나 해시값에 전자서명하여 MAC과 동일한 효과를 얻을 수 있다. 전자서명을 사용하면 연산 비용이 비싼 측면이 있지만, MAC과 달리 누구나 확인 가능하며, 부인방지를 제공할 수 있다는 장점도 있다.

2.1 대칭 암호알고리즘을 이용한 MAC

MAC과 대칭 암호알고리즘은 모두 메시지와 키를 입력받는다. 하지만 대칭 암호알고리즘으로 메시지를 암호화한 출력의 크기는 입력 크기에 비례한다. 따라서 대칭 암호알고리즘을 이용하여 MAC을 만들기 위해서는 암호화한 결과로부터 일정한 크기의 출력을 계산할 수 있어야 한다. 이를 위해 CBC 모드로 메시지를 암호화하고 마지막 블록을 MAC 값으로 사용하는 방법을 고려할 수 있다. CBC 모드로 메시지를 암호화하였을 때 마지막 블록은 모든 평문 블록에 영향을 받기 때문에 MAC 값 역할을 충분히 할 수 있다.

하지만 이 방법은 위조가 가능하기 때문에 마지막 암호 블록을 직접 MAC 값으로 사용할 수는 없다. 위조 문제를 극복하기 위해 메시지 길이가 l일 때 $K_l = E.K(l)$을 계산하여 MAC 키인 K 대신에 K_l을 이용하여 메시지를 CBC 모드로 암호화하면 추가적인 조치 없이 마지막 블록을 MAC 값으로 사용할 수 있다.

표준으로 정의된 CBC-MAC은 두 개의 키가 필요하다. 이를 K_1과 K_2라 하자. CBC-MAC은 키 K_1을 이용하여 메시지를 CBC 모드로 암호화한 다음 결과 암호문의 마지막 블록을 키 K_2를 이용하여 다시 암호화하여 만들어진 암호문을 MAC 값으로 사용한다. 이와 같이 계산하는 MAC을 종종 ECBC-MAC(Encrypt-last-block CBC-MAC)이라 한다. 여기서 IV는 모두 0bit를 사용하며, 채우기는 비트 채우기를 사용한다. MAC에서 필요한 채우기와 대칭암호에서 필요한 채우기의 요구사항이 다르다. 후자는 복호화 후에 정확하게 제거할 수 있어야 하는 것이 필요하지만 전자는 이것이 필요 없고, 같은 메시지에 대해 같은 MAC 값을 얻기 위해서는 반드시 동일한 채우기 방법을 사용해야 한다. IV를 항상 모두 0bit를 사용하는 이유도 같은 이유이다.

MAC도 해시함수와 마찬가지로 충돌회피 특성을 제공해야 한다. MAC 값의 출력이 n비트이면 생일 파라독스 때문에 $2^{n/2}$개 MAC 값을 계산하면 충돌을 찾을 수 있다. AES를 이용하여 CBC-MAC을 만들면 MAC 값의 길이가 128bit이므로 2^{64}개의 MAC 값을 계산하면 충돌을 찾을 수 있게 된다. 따라서 충돌을 찾을 확률[2]을 $1/2^{32}$ 수준으로 유지하고자 하면 2^{48}개의 MAC 값을 계산한 후에는 반드시 키를 바꾸어야 한다.

CMAC은 CBC-MAC의 변형으로 3개의 키(K, K_1, K_2)를 사용하지만 3개를 랜덤하게 생성하지 않고 K_i는 K를 이용하여 생성한다. 따라서 하나의 키만 사용하는 MAC이라는 의미에서 OMAC(One-key MAC)이라고도 한다. CMAC은 메시지를 CBC 모드로 암호화를 하는데, 마지막 평문 블록 M_n만 일반 CBC 암호화 달리 다음과 같은 처리를 한 다음 암호화한다.

- M_n이 완전 블록인 경우: $M'_n = M_n \oplus K_1$
- M_n이 완전 블록이 아닌 경우: $M'_n = (M_n || 10\cdots0) \oplus K_2$

CMAC은 이처럼 채우기를 항상 하지 않으며, 채우기가 필요할 경우에도 비트 채우기를 한다. CMAC은 이렇게 하여 얻은 마지막 블록을 MAC 값으로 활용한다.

NMAC은 두 개의 키를 사용하는데 이 키를 암호알고리즘의 대칭키로 사용하지 않고, 암호화하는 평문으로 사용한다. NMAC은 먼저 메시지를 사용하는 블록 암호알고리즘 키 길이로 나누어 각 나눈 것을 대칭키로 사용하여 K_1을 연속적으로 암호화한다. 이것을 식으로 표현하면 다음과 같다.

$$\begin{aligned} C_1 &= E.M_1(K_1) \\ C_2 &= E.M_2(C_1) \\ \vdots &\quad \vdots \\ C_n &= E.M_n(C_{n-1}) \end{aligned}$$

이렇게 하여 얻은 C_n를 채우기하여 최종 대칭키를 만들고, 이 키를 이용하여 K_2를 암호화하여 최종 MAC 값을 계산한다. 이 과정에서 블록 크기와 대칭키 길이가 같으면 채우기가 필요없다. NMAC에서 사용하는 두 개의 키 길이는 사용하는 블록 암호알고리즘의 블록 길이와 같아야 한다.

NMAC은 메시지 블록을 키로 사용하기 때문에 대칭 암호알고리즘에 사용하는 키가 MAC을 계산하는 동안 계속 바뀐다. 따라서 내부적으로 사용하는 라운드키를 매번 새롭게 계산해야 하기 때문에 성능은 CBC-MAC, CMAC이 더 우수하다.

[2]생일 파라독스에 의한 충돌을 찾을 확률은 대략 $\frac{q^2}{|X|}$이며, 여기서 q는 생성한 개수이고, $|X|$는 값의 길이이다. 따라서 q가 2^{48}이 되면 충돌을 찾을 확률은 $\frac{1}{2^{32}}$ 정도가 된다.

PMAC은 기존 CBC-MAC, NMAC 등은 일련의 순차적인 암호화를 통해 MAC 값을 계산하는 문제점을 극복하고자 개발된 MAC이다. PMAC은 MAC 값을 계산하기 위해 순차적으로 암호화하지 않고 병행으로 암호화할 수 있다. 이것을 식으로 표현하면 다음과 같다.

$$\begin{aligned} C_1 &= E.K(M_1 \oplus g(K_1, 1)) \\ C_2 &= E.K(M_2 \oplus g(K_1, 2)) \\ \vdots &\quad \vdots \\ C_{n-1} &= E.K(M_{n-1} \oplus g(K_1, n-1)) \end{aligned}$$

마지막 블록이 완전 블록이면 $g(K_1, n)$으로 XOR하여 C_n으로 사용하고, 완전 블록이 아니면 비트 채우기를 하여 그 자체를 C_n으로 사용한다. PMAC은 이렇게 구한 암호문 블록을 모두 XOR한 다음, 이 값을 다시 K를 이용하여 암호화하여 얻은 암호문을 MAC 값으로 사용한다. PMAC은 일부 블록만 변경되면 기존 값을 이용하여 빠르게 새 MAC 값을 계산할 수 있기 때문에 점진적(incremental)으로 MAC 값을 계산할 수 있다고 한다. PMAC은 다중 프로세서를 활용할 수 있다는 이점은 있지만 같은 키로 암호화하는 메시지 길이도 안전성에 영향을 주기 때문에 CBC-MAC이나 NMAC보다 키를 더 자주 바꾸어야 한다.

MAC 중에 키를 한 번만 사용할 수 있는 MAC이 있으며, 이와 같은 MAC을 일회용 MAC(one-time mac)이라 한다. 일회용 MAC은 CBC-MAC, CMAC, NMAC, PMAC 보다 더 효과적으로 구성할 수 있지만 한 번만 사용할 수 있기 때문에 실용적으로 사용하는 데 문제가 있다. 하지만 Carter와 Wegman은 일회용 MAC을 이용하여 여러 번 사용할 수 있는 MAC을 구성하는 안전한 방법을 제시하였으며[10], 이 형태의 MAC을 Carter-Wegman MAC이라 한다. Carter-Wegman MAC은 다음과 같이 표현할 수 있다.

$$CW(K_1, K_2, M) = (r, E.K_1(r) \oplus S.K_2(M))$$

여기서 S는 일회용 MAC이다. 이전 장에서 살펴본 GCM 모드에서 사용한 MAC이 이 형태의 MAC이다.

2.2 HMAC

HMAC은 충돌회피 해시함수를 이용하는 MAC 함수를 말한다. SHA-1과 SHA-2처럼 메시지를 블록으로 나누어 동일 과정을 반복적으로 적용하여 메시지를 압축하는 Merkle과 Damgård 구조의 해시함수를 사용할 경우 단순하게 MAC 함수를 만들면 안전하지 못할 수 있다. 예를 들어 $H(K||M)$ 형태로 MAC을 계산하면 해당 MAC값을 이용하여 $H(K||M||M')$을 계산할 수 있는 심각한 문제가 있어 안전하지 못하다. $H(M||K)$ 형태로 하는 것도 H의 충돌을 찾은 공격자는 MAC의 충돌을 찾을 수 있어서 이 역시 안전하지 못하다. 따라서 $H(K_1||M||K_2)$나 $H(K_1||H(K_2||M))$ 형태로 MAC을 계산하는 것이 더 안전한 방법이다. 실제 인터넷 표준[11]에서는 $H(K_1||H(K_2||M))$ 형태를 사용하고 있다. 이 표준의 구체적인 계산 방법은 아래와 같다.

$$\text{MAC}.K(M) = H(K' \oplus \text{opad}||H(K' \oplus \text{ipad}||M))$$

여기서 K'은 K에 0 채우기를 하여 해시함수의 블록크기와 같도록 만든 키 값이다. 또 opad와 ipad는 서로 다른 상수 비트 문자열이며, 이 둘은 하나의 키로부터 두 개의 다른 키를 만들기 위한 요소로 생각하면 된다. 즉, $K_1 = K0...0||\text{opad}$, $K_2 = K0...0||\text{ipad}$라 하면 $\text{MAC}.K(M) = H(K_1||H(K_2||M)$이 되며, $H(K_1)$과 $H(K_2)$는 사전계산하여 사용할 수 있다.

SHA-3는 Merkle과 Damgård 구조가 아니므로 $H(K||M)$ 형태로 MAC을 계산하여도 문제가 없지만 $H(M||K)$ 형태는 여전히 안전하지 않다.

3. 공개키 암호알고리즘

3.1 RSA

RSA는 Rivest, Shamir, Adleman이 1977년에 개발한 공개키 암호알고리즘이다[12]. 이 알고리즘은 현재 가장 널리 사용하고 있는 암호알고리즘 중 하나이다. 이 알고리즘은 인수분해 문제의 어려움에 기반을 두고 있다. RSA의 공개키 쌍은 다음과 같은 단계를 이용하여 생성한다.

- 단계 1. 매우 큰 같은 크기의 소수 2개 p와 q를 선택한다.
- 단계 2. $n = pq$를 계산한다.
- 단계 3. $\phi(n) = (p-1)(q-1)$를 계산한다.
- 단계 4. $\gcd(\phi(n), e) = 1$인 $\phi(n)$보다 작은 e를 임의로 선택한다.
- 단계 5. $ed \equiv 1 \pmod{\phi(n)}$인 d를 계산한다.

이 과정에서 얻은 값 중 사용자의 공개키는 (n, e)가 되며, 개인키는 (n, d)가 된다. 단계 1에서 소수는 소수 검사(primality test)를 이용하여 생성한다. 임의의 수를 생성한 후 해당 값이 소수인지 아닌지를 검사하는 방식으로 생성한다. 현재 n은 2,048bit 정도 되어야 안전하다고 하므로 p와 q는 1,024bit 정도 되어야 한다. 단계 3에서 $\phi(n)$ 대신에 $\lambda(n) = (lcm)(p-1, q-1)$를 이용할 수 있으며, 이 경우 d 값이 작아져 효율성을 높일 수 있다. 또 e로 $65,537 = 2^{16} + 1$을 많이 사용한다. 이것은 제곱-곱하기(square-multiply) 방법을 사용하여 암호화하기 때문이다. 제곱 곱하기 방식은 2^{33}을 32번의 곱셈 연산을 이용하여 계산하는 것이 아니라 2×2, $2^2 \times 2^2$, $2^4 \times 2^4$, $2^8 \times 2^8$, $2^{16} \times 2^{16}$, $2^{32} \times 2$와 같이 곱셈을 6번만 하여 계산하는 방식을 말한다. 이처럼 e로 3이나 65,537를 많이 사용하기 때문에 RSA는 보통 공개키로 암호화하는 속도가 상대적으로 개인키로 복호화하는 속도에 비해 빠르다.

RSA에서 암호화와 복호화는 다음과 같다.

$$\begin{aligned} C &= M^e \mod n \\ M &= C^d \mod n \end{aligned}$$

이것이 성립하는 이유는 다음과 같다.

$$C^d = (M^e)^d = M^{ed} = M^{k\phi(n)+1} = \left(M^{\phi(n)}\right)^k M \equiv M \pmod{n}$$

물론 M이 n과 서로소가 아니면 이 식이 성립하지 않는다. 하지만 M이 n과 서로소가 아니더라도 M은 q와 서로소이거나 p와 서로소이기 때문에 M이 n과 서로소가 아닌 경우에는 이를 이용하여 RSA의 정확성을 증명할 수 있다.

실제 RSA는 안전성 문제 때문에 위와 같이 암호화하지 않는다. 예를 들어 C_1이 M_1의 암호문이고, C_2가 M_2의 암호문이면 C_1C_2는 M_1M_2의 암호문이 된다. 또한 속도를 위해 3과 같은 작은 지수를 e로 사용할 수 있는데, 이 경우에는 평문의 특성에 따라 전수조사를 통해 M을 구할 수 있다. 따라서 표준에서는 OAEP(Optimal Asymmetric Encryption Padding)[13]을 사용하여 메시지를 더 랜덤한 값으로 바꾸고 그 길이를 RSA 법과 같은 길이로 확장하도록 하고 있다. OAEP는 메시지를 암호화하기 전에 다음 두 값을 계산하게 된다.

$$\begin{aligned} s &= (M||00\cdots0) \oplus G(r) \\ t &= H(s) \oplus r \end{aligned}$$

여기서 G와 H는 해시함수이며, r은 사용자가 임의로 선택한 랜덤수이다. 이렇게 하여 얻은 s와 t를 비트 결합하여 RSA 암호화를 한다. 결과적으로 암호화하는 값의 길이를 n의 길이와 유사하게 만드는 것이 목표이다.

r의 길이와 채우기의 길이는 사용하는 시스템에서 결정하여 고정하며, s를 계산하기 위한 $(M||00\cdots0)$ 부분은 시스템에 따라 차이가 있을 수 있다. G와 H를 모두 SHA-256을 이용한다고 가정하면 $G(r)$의 길이는 SHA-256의 출력 길이인 32byte보다 훨씬 커야 한다. 이를 위해 $G(r)||G(r+1)||\cdots$ 형태로 필요한 만큼 확장하여 사용하면 된다.

OAEP를 이용하여 암호화된 메시지를 복호화하면 s와 t를 얻을 수 있고, 다음 식을 이용하여 이 두 개의 값으로부터 M을 계산할 수 있다.

$$\begin{aligned} r &= t \oplus H(s) \\ M &= s \oplus G(r) \end{aligned}$$

OAEP 대신에 혼합 암호를 이용하는 경우도 많다. 이를 RSA-KEM(Key Encapsulation Mechanism)이라 하며, 랜덤 대칭키를 OAEP를 이용하여 암호화하지 않고, 상대방의 공개키가 (n, e)일 때 랜덤한 $x \in_R \mathbb{Z}_n^*$를 암호화하고, 이 x를 KDF에 입력하여 대칭키를 생성하여 다음과 같이 메시지 M을 암호화한다.

$$x \in_R \mathbb{Z}_n^*, K_1||K_2 = \mathrm{KDF}(x), c = x^e \mod n, C = \{M\}.K_1, \mathrm{MAC}.K_2(C)$$

3.2 ElGamal

ElGamal 공개키 암호알고리즘[14]은 이산대수 문제에 기반한 확률 암호알고리즘이다. ElGamal 공개키 방식은 이산대수 문제에 기반하기 때문에 이산대수 문제가 어려운 군의 생성이 선행되어야 한다. 보통 매우 큰 소수 p를 선택하여 $\mathbb{Z}_p^*$ 군을 활용하기도 하고, 이 군의 위수가 소수 q인 부분군을 사용하기도 한다. 현재 p는 2,048bit 정도 되어야 하며, q는 224bit 정도 되어야 안전한다고 한다.

ElGamal에서 $\mathbb{Z}_p^*$의 위수가 소수 q인 부분군은 보통 다음과 같이 생성한다.

- 단계 1. 원하는 크기의 소수 q를 선택한다.
- 단계 2. $p = kq + 1$를 계산 한 후에 p가 소수인지 판단한다. 아니면 이 과정을 반복한다. 여기서 k는 랜덤한 수로 원하는 크기의 p를 얻기 위해 사용하는 수이다.
- 단계 3. p보다 작은 수 h를 임의로 선택하여 $g = h^k \mod p$를 계산한다. 이 값이 1이 아니면 G_q의 생성자이다. 만약 1이면 이 과정을 다시 반복한다.

여기서 p, q, g를 군 파라미터라 한다. 사용자의 개인키 $x \in \mathbb{Z}_q^*$는 임의로 선택하며, 대응되는 공개키는 $y = g^x \mod p$가 된다.

이와 같이 랜덤하게 군을 생성할 경우 약한 군을 사용할 확률이 있고, 군을 설정하는 측에서 트랩도어 목적으로 일부러 약한 군을 사용할 수 있는 문제점도 있다. 이에 RFC 7919[15]와 같은 표준에 제시된 군을 사용하는 것이 더 안전하다. 물론 이전 RFC 5114[16]에 제시된 군 중 나중에 문제가 있는 군으로 밝혀진 경우도 있다. 이 때문에 일반 유한체(finite field) 기반 군을 사용하지 않고 타원곡선을 이용하는 경향이 높아지고 있다.

ElGamal에서 사용자 A의 공개키 $y_A = g^{x_A} \mod p$를 이용한 메시지 M의 암호화는 다음과 같다.

$$(C_1, C_2) = (g^r, y_A^r M)$$

<표 9.1> 권장하는 공개키의 길이

기간	안전성	대칭 암호	RSA 법 n	이산대수		타원곡선	해시함수	HMAC
				q	p			
2019–2030	112	AES-128	2,048	224	2,048	224	SHA-224	
2019–2030 이후	128	AES-128	3,072	256	3,072	256	SHA3-256	SHA-1
2019–2030 이후	192	AES-192	7,680	384	7,680	384	SHA3-384	SHA3-224
2019–2030 이후	256	AES-256	15,360	512	15,360	512	SHA3-512	SHA3-256 이상

여기서 r는 $\mathbb{Z}_q^*$에서 임의로 선택한 랜덤 수이며, 메시지 M은 G_q의 원소이어야 한다. 또한 mod p가 식에서 생략되어 있다. 위 암호문의 복호화하는 다음과 같다.

$$M = C_2(C_1^{x_A})^{-1} = y_A^r M(g^{rx_A})^{-1} = g^{x_A r} g^{-rx_A} M = M$$

일반적인 데이터를 암호화하고 싶으면 다음과 같은 혼합 암호(hybrid encryption)를 사용한다.

$$(C_1, C_2, \text{tag}) = (g^r, C = E.K_1(M), T = \text{MAC}.K_2(C))$$

여기서 E는 대칭 암호알고리즘이며, K_1과 K_2는 y_A^r을 KDF에 입력하여 생성한 대칭키이다. 수신자는 $C_1^{x_A}$을 이용하여 K_1과 K_2를 계산한 후에 이를 이용하여 인증 암호화를 복호화한다.

3.3 공개키의 길이

컴퓨팅 성능이 계속 발전함에 따라 인수분해와 이산대수 문제를 해결하는데 걸리는 시간도 점점 감소하고 있다. 이 때문에 RSA에서 사용하는 법 n의 크기와 ElGamal에서 사용하는 이산대수 군의 크기도 계속 늘려야 하는 문제점이 있다. 2020년 미국 표준기구인 NIST에서 권장하는 키 길이는 표 9.1과 같다. 이 표에서 알 수 있듯이 키 길이 문제 때문에 일반 이산대수 대신에 점점 타원곡선 기반 공개키 암호알고리즘이나 전자서명 알고리즘을 사용하는 응용이 많아지고 있다. 이 표에 제시된 RSA, 이산대수, 타원곡성의 안전성은 양자 컴퓨팅이 현실화되면 아무 의미가 없어진다.

4. 전자서명

4.1 RSA 전자서명

RSA 암호알고리즘은 공개키는 물론 개인키로도 메시지를 암호화할 수 있다. 개인키로 메시지를 암호화하게 되면 이것은 오직 개인키를 가지고 있는 사용자만 할 수 있으므로 전자서명 역할을 할 수 있다. 하지만 단순하게 개인키로 암호화하면 위조가 가능하기 때문에 충돌회피 해시함수를 보통 사용한다. 예를 들어 $M = 1$이면 아무 사용자의 전자서명을 만들어 낼 수 있다. 또한 두 개의 서명 σ_1과 σ_2를 곱하여 $\sigma_1\sigma_2$에 대한 전자서명을 만들어 낼 수 있다. 따라서 $H(M)^d \mod n$ 형태로 계산하여야 안전하다. 하지만 이 경우에도 $H(M)$의 값이 n에 비해 매우 작은 값이므로 안전성을 높이기 위해 암호화할 때 OAEP를 사용한 것처럼 해시값의 길이가 n의 길이와 같도록 확장하여 서명한다. 이것을 FDH(Full Domain Hash)라 한다. Bellare와 Rogaway는 결정적 RSA 서명을 확률적으로 바꾸는 동시에 서명할 값의 길이를 n의 길이와 같아지도록 하는 RSA-PSS(Probabilistic Signature Scheme)[17]을 제안하였다. 이 방식에서 메시지 M에 대한 서명 σ는 다음과 같이 계산된다.

- 단계 1. $r \in_R \{0,1\}^{k_0}$를 임의로 선택한다.
- 단계 2. $w = H(M||r)$를 계산한다.
- 단계 3. $M' = 0||w||(G_1(w) \oplus r)||G_2(r)$를 계산한다.
- 단계 4. $\sigma = M'^d \mod n$을 계산한다.

여기서 k_0과 H 출력값의 길이는 모두 256bit라 하고, n은 2,048bit라 하자. 그러면 G_1도 256bit이고, G_2는 1,535bit가 되어야 한다. 따라서 OAEP에서처럼 G_2는 주어진 입력을 확장해 주어야 한다.

RSA-PSS에서 서명 (M, σ)의 확인은 다음과 같이 진행된다.

- 단계 1. $M' = \sigma^e \mod n$를 계산한다.
- 단계 2. $M' = b||w||\alpha||\gamma$로 나눈다.
- 단계 3. $r = \alpha \oplus G_1(w)$를 계산한다.
- 단계 4. $b \stackrel{?}{=} 0$, $G_2(r) \stackrel{?}{=} \gamma$, $H(M||r) \stackrel{?}{=} w$를 확인한다.

4.2 DSA 전자서명

이산대수 기반 전자서명 중 가장 간결하고 효과적인 안전한 기법 중 하나가 Schnorr 전자서명 기법이다[18]. 하지만 이 기법은 특허 문제로 표준으로 제정할 수 없었다. 이것은 RSA도 마찬가지이다. 이 때문에 Schnorr 전자서명을 약간 변형한 기법을 표준으로 제정하였으며, 이 표준을 DSA라 한다. DSA는 1991년에 전자서명 표준에 사용하기 위해 처음 제안되었고, 1994년에 FIPS 186으로 공식 제정되었다. 그 이후 4번의 수정이 있었다. 가장 마지막 버전이 2013년에 발표된 FIPS 186-4이다[19]. 현재 FIPS 186-5 초안에 의하면 FIPS 186-4 버전의 DSA는 폐기되고 새 전자서명 표준으로 바뀔 예정이며, 새 표준은 Schnorr 전자서명의 특허 만료로 Schnorr 전자서명과 매우 가깝게 바뀔 예정이다.

DSA는 이산대수 문제에 기반하고 있으며, 사용자 공개키 쌍의 생성은 ElGamal 암호알고리즘과 차이가 없다

순환군 파라미터가 p, q, g이고, 사용자 A의 개인키 $x_A \in \mathbb{Z}_q^*$이며 공개키는 $y_A = g^{x_A} \mod p$일 때, 메시지 M에 대한 ElGamal 전자서명은 다음과 같이 계산한다.

- 단계 1. $w \in_R \mathbb{Z}_q^*$를 임의로 선택한다.
- 단계 2. $W = (g^w \mod p) \mod q$를 계산한다.
- 단계 3. $s = (H_q(M) + x_A W)w^{-1} \mod q$를 계산한다.

결과 서명은 M, W, s이다. 사용자 A의 공개키를 가지고 있는 사용자는 다음 단계를 통해 서명을 확인한다.

- 단계 1. $u_1 = H_q(M)s^{-1} \mod q$를 계산한다.
- 단계 2. $u_2 = Ws^{-1} \mod q$를 계산한다.
- 단계 3. $W? \equiv (g^{u_1} y^{u_2} \mod p) \pmod q$인지 확인한다.

DSA 서명의 정확성은 다음과 같이 확인할 수 있다.

$$W = g^w \quad = \quad g^{u_1} y^{u_2} = g^{H_q(m)s^{-1}} g^{x_A W s^{-1}} = g^{s^{-1}(H_q(m)+x_A W)} = g^{(H_q(m)+x_A W)^{-1} w (H_q(m)+x_A W)} = g^w$$

4.3 ECDSA 전자서명

현재는 DSA 전자서명보다 타원곡선 기반 전자서명을 더 많이 사용한다. ECDSA는 DSA의 타원곡선 버전으로 2000년에 FIPS 186-2에 처음 포함되었다. 타원곡선을 이용하여 유한순환군을 만들 수 있으므로 기존 이산대수 기반 알고리즘이나 프로토콜을 그대로 타원곡선으로 옮길 수 있다. 수학식으로 보면 g^a가 aP 형태로 바뀔 뿐이다. 여기서 P는 타원곡선 위의 점이며, 타원곡선을 이용한 군은 곱셈군이 아니라 덧셈군이다. 따라서 aP는 P를 a번 거듭 더한 좌표(예: $3P = P + P + P$)이며, 군은 닫혀 있으므로 aP는 사용하는 타원곡선 위의 또 다른 점이다.

P가 위수가 소수 q인 부분군의 생성자이고, $d_A \in \mathbb{Z}_q^*$가 서명자의 서명키이며, $Q_A = d_A P$가 확인키이면 DSA 전자서명은 다음과 같이 계산한다.

- 단계 1. $w \in_R \mathbb{Z}_q^*$를 임의로 선택한다.
- 단계 2. $(a, b) = wP$를 계산한다. 여기서 a는 점 wP의 x 좌표이다.
- 단계 3. $s = (H_q(M) + d_A a)w^{-1} \mod q$를 계산한다.

결과 서명은 M, a, s이다. 사용자 A의 공개키를 가지고 있는 사용자는 다음 단계를 통해 서명을 확인한다.

- 단계 1. $(\hat{a}, \hat{b}) = H_q(M)s^{-1}P + as^{-1}Q_A$를 계산한다.
- 단계 2. $a? \equiv \hat{a} \pmod q$인지 확인한다.

Schnorr 전자서명의 특허가 만료됨에 따라 효율적이며 안전성이 증명된 Schnorr 전자서명을 표준으로 채택하기 위한 노력이 진행되었고, 그것의 결과 중 하나가 EdDSA이다[20]. EdDSA는 타원곡선 기반이며, 기존에 많이 사용한 타원곡선 대신에 Edwards25519라는 곡선을 사용한다. 이 곡선은 기존에 많이 사용하고 있는 Curve25519 보다 더 빠르게 연산을 수행할 수 있다,

EdDSA는 대칭키처럼 128비트 랜덤한 값을 개인키 K로 사용한다. 이 키를 SHA-512에 입력하여 두 개의 256 비트 키 x와 r를 얻는다. 이 사용자의 공개키는 $Q = xP$가 된다. ECDSA와 마찬가지로 P는 위수가 소수 q인 부분군의 생성자이다. EdDSA에서 메시지 M에 대한 서명은 다음과 같이 계산한다.

- 단계 1. $w = H(r||M) \mod q$를 계산한다.
- 단계 2. $W = wP$를 계산한다.
- 단계 3. $c = H(W||Q||M) \mod q$를 계산한다.
- 단계 4. $s = w + cx \mod q$를 계산한다.

결과 서명은 M, W, s이다. 공개키 Q를 가지고 있는 사용자는 다음 단계를 통해 서명을 확인한다.

- 단계 1. $c = H(W||Q||M) \mod q$를 계산한다.
- 단계 2. $sP? \equiv W + cQ \pmod q$인지 확인한다.

Schnorr 전자서명에서는 매번 다른 w를 사용해야 한다. 이를 위해 EdDSA는 w를 랜덤하게 선택하지 않고 키의 일부와 서명할 메시지를 이용하여 계산한다.

참고문헌

[1] I. Damgård, "A Design Principle for Hash Functions," In Advances in Cryptology, CRYPTO '89, LNCS 435, Springer, pp. 416–427, 1989.

[2] R. C. Merkle, "One Way Hash Functions and DES," In Advances in Cryptology, CRYPTO '89, LNCS 435, Springer, pp. 428–446, 1989.

[3] A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996.

[4] FIPS, Secure Hash Standard, Federal Information Processing FIPS 180-1, 1995.

[5] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, "Finding Collisions in the Full SHA-1," Advances in Cryptology, Crypto 2005, LNCS 3621, pp. 17–36, Springer, 2005.

[6] Marc Stevens, "New Collision Attacks on SHA-1 based on Optimal Joint Local-Collision Analysis," Advances in Cryptology, EUROCRYPT 2013, LNCS 7881, pp. 245–261, Springer, 2013.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, Giles Van Assche, "The KECCAK SHA-3 Submission", `https://keccak.team/files/Keccak-submission-3.pdf`, Jan. 2011.

[8] Tetsu Iwata, Kaoru Kurosawa "OMAC: One-Key CBC MAC," Int'l Workshop on Fast Software Encryption, FSE 2003, LNCS 2887, pp 129–153, Springer, 2003.

[9] John Black, Phillip Rogaway, "A Block-Cipher Mode of Operation for Parallelizable Message Authentication," Advances in Cryptology, EUROCRYPT 2002, LNCS 2332, pp. 384–397, Springer, 2002.

[10] M. Wegman, L. Carter, "New Hash Functions and Their Use in Authentication and Set Equality," Journal of Computer and System Sciences, Vol. 22, pp. 265–279, 1981.

[11] Mihir Bellare, Ran Canetti, Hugo Krawczyk, "HMAC: Keyed-Hashing for Message Authentication," IETF RFC 2104, Feb. 1997.

[12] R. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, Vol. 21, No. 2, pp. 120–126, Feb. 1978.

[13] Mihir Bellare, Phillip Rogaway, "Optimal Asymmetric Encryption - How to encrypt with RSA," Advances in Cryptology, EUROCRYPT 1994, LNCS 950, pp. 91–121, Springer, 1995.

[14] Taher ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," IEEE Transactions on Information Theory, Vol. 31, No. 4, pp. 469–472, Jul. 1985.

[15] D. Gillmor, "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)," IETF RFC 7919, Aug. 2016.

[16] M. Lepinski, S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards," IETF RFC 5114, Jan. 2008.

[17] Mihir Bellare, Phillip Rogaway, "The Exact Security of Digital Signatures - How to Sign with RSA and Rabin," Advances in Cryptology, EUROCRYPT 1996, LNCS 1070, pp. 399–416, Springer, 1996.

[18] Claus-Peter Schnorr, "Efficient Signature Generation by Smart Cards," J. of Cryptology, Vol. 4, No. 3, pp. 161–174, 1991.

[19] FIPS, Digital Signature Standard (DSS), Federal Information Processing Standard FIPS 186-4, 2013.

[20] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, "High-speed High-security Signatures," J. of Cryptographic Engineering, Vol. 2, No. 77–89, Springer, Aug. 2012.

퀴즈

1. $p = 53$, $q = 59$를 선택하여 $n = pq = 3,127$과 $\phi(n) = 3,016$을 계산하였다. 그다음 $e = 3$을 선택하였고, $d = 2,011$을 계산하여 RSA 공개키 쌍을 생성하였다. $m(< n)$에 대한 RSA 전자서명 값을 $m^d \mod n$을 통해 계산한다고 가정하자. 그러면 $m_1 = 1,000$의 서명값은 10이고, $m_2 = 3$의 서명값은 934이다. 그러면 $m_3 = 3,000$의 서명값은?

2. NMAC은 메시지를 키 길이로 나눈 값을 대칭키로 사용하여 MAC 키를 연속적으로 암호화를 한다. NMAC도 이렇게 암호화한 결과를 그대로 사용하면 위조가 가능하다. 마지막에 추가적인 조처 없이 그대로 사용한다고 가정하고, 사용하는 대칭 암호알고리즘의 키 길이와 길이가 같은 메시지 X에 대한 NMAC이 T이고 Y도 X와 길이가 같은 메시지이면 $X||Y$에 대한 MAC값은? 단, X는 정확하게 블록 크기의 배수이며 T는 X에 채우기 없이 사용하여 만든 값이라고 가정한다.

   ① $E.Y(T)$
   ② 알 수 없음
   ③ $E.K(Y)$
   ④ $E.K(X)$

3. CBC 모드로 메시지를 암호화하였을 때 마지막 암호문 블록은 메시지 전체에 영향을 받기 때문에 MAC 값으로 활용할 수 있는 특성을 충분히 갖추고 있다. 하지만 마지막 암호문 블록을 그대로 사용하면 위조가 가능하기 때문에 추가 조처가 필요하다. 추가 조처 없이 마지막 암호문 블록을 그대로 사용한다고 가정하자. 모든 비트가 0인 IV를 사용하고, X는 사용하는 대칭 암호알고리즘 블록 크기의 데이터일 때, MAC.$K(X) = T$이면 $X||X \oplus T$의 MAC값은? 참고로 CBC-MAC은 마지막 블록이 완전 블록이면 채우기를 하지 않고 부족하면 비트 채우기를 한다.

   ① 알 수 없음
   ② X
   ③ 모든 비트가 0
   ④ T

연습문제

1. SHA 계열의 해시함수나 CBC-MAC 등은 대칭암호알고리즘과 마찬가지로 채우기가 필요하다. 채우기가 필요한 이유와 보안적으로 그것의 중요한 이유(모두 0으로 채우기를 하거나 비트 채우기만 하였을 때 문제점을 제시)를 설명하시오. 또 대칭 암호알고리즘의 채우기와 중요한 차이점을 설명하시오. 힌트. SHA 계열은 Merkle과 Damgård 구조 기반이기 때문에 내부적으로 블록 단위의 함수를 활용한다.

2. $MAC.K(M) = H(K||M)$ 형태로 사용할 경우의 문제점을 설명하시오.

3. RSA 암호알고리즘에서 공개키가 (n, e)이고 개인키가 (n, d)일 때, $M < n$인 메시지의 암호화는 $C = M^e \mod n$을 이용한다. 이 암호문은 $C^d \mod n$을 통해 복호화한다. 복호화 결과의 정확성은 $\gcd(M, n) = 1$일 때, 다음을 통해 증명할 수 있다.

$$C^d = M^{ed} = M^{k\phi(n)+1} \equiv \left(M^{\phi(n)}\right)^k M \equiv M \pmod{n}$$

   참고. 오일러 정리. $\gcd(m, n) = 1$이면 $m^{\phi(n)} \equiv 1 \pmod{n}$이 성립한다.

   $\gcd(M, n) \neq 1$이면 M은 p의 배수이거나 q의 배수이다. 따라서 M은 p와 서로소이거나 q와 서로소이다. $M = ap$이고 q와 서로소라고 가정하면 다음이 성립한다.

$$M\left(M^{\phi(n)}\right)^k \equiv M\left(M^{\phi(q)}\right)^{\phi(p)k}$$

   M은 q와 서로소이기 때문에 $M^{\phi(q)} \equiv 1 \pmod{q}$이다. 따라서 다음이 성립한다.

$$M\left(M^{\phi(q)}\right)^{\phi(p)k} = ap(1 + bq)$$

   $ap(1 + bq) \equiv M \pmod{n}$임을 보이시오. 이것이 성립하면 모든 $M < n$에 대해 RSA의 정확성이 증명된다.

4. RSA 암호알고리즘은 공개키를 효율성 때문에 종종 공개키 e 값으로 3을 선택하여 사용하는 경우가 있다. $e = 3$이고 $n > 30$일 때, 대응되는 d로 암호화된 값을 임의로 하나 만들어 보시오. $c^e \equiv m \mod n$가 성립하는 m, c 쌍을 제시하시오.

5. RSA 공개키가 (n, e)인 사용자에게 1부터 365 사이의 수 m을 암호화하여 전달하고자 한다. 이를 위해 매우 큰 수 x를 선택한 후에 x와 $(m + x)^e \mod n$를 전달하였다. 이 방식의 문제점을 제시하시오.

6. $m(< n)$에 대한 RSA 전자서명 값을 $m^d \mod n$을 통해 계산한다고 가정하자. 서명자의 서명키가 (n, d)이고, 확인키가 (n, e)일 때, $m = r^e m' \mod n$을 계산하여 서명자로부터 m에 대한 서명값 $m^d \mod n$을 확보하였다. 이 서명으로부터 서명자의 또다른 유효한 서명을 만들 수 있다. 해당 서명을 얻는 방법을 제시하시오.