# 정보보호개론

제11장 해시함수의 활용

## 1. 해시함수

해시함수는 임의 길이의 입력을 받아 그것을 대표할 수 있는 고정된 짧은 길이의 값을 출력하여 주는 결정적 함수이다. 여기서 짧은 길이란 생일 파라독스를 고려한 충돌회피에 안전한 길이를 말한다. 현재는 보통 256bit 이상의 길이를 사용하고 있다. 1장부터 소개한 바와 같이 해시함수는 전자서명, 무결성, 비트 약속 등에 활용하고 있다.

전자서명에서는 효율성과 안전성을 높이기 위해 메시지에 직접 서명하는 대신에 보통 메시지 해시값에 서명한다. RSA 전자서명의 경우 해시값에 서명하면 서명하는 메시지의 크기와 상관없이 서명하는 값의 크기가 고정되어 효율적이다. 또한 작은 값의 메시지에 직접 개인키로 암호화하는 방식은 위조가 가능하지만 해시값을 사용하면 이것을 방어할 수 있다. 이때 안전성을 높이기 위해 서명하는 값의 길이가 RSA 법의 길이와 같아지도록 하는 것이 바람직하다. 이처럼 해시함수의 출력 크기 대신에 특정 크기의 출력을 얻도록 확장하는 것을 전체 영역 해시라 한다. 보통 해시함수는 카운터를 이용하여 필요한 만큼의 길이로 다음과 같이 확장할 수 있다.

$$H(M||1)||H(M||2)||\cdots||H(M||k)||\cdots$$

전자서명에서 활용하는 해시함수는 반드시 충돌회피 해시함수이어야 한다. RSA의 경우 위와 같은 방법보다는 랜덤 요소를 포함하는 RSA-PSS 방식을 사용하며, 이 방식을 사용하면 부가적으로 결정적인 RSA 알고리즘이 확률적 알고리즘으로 바뀌는 효과도 있다. RSA의 경우 공개키로 암호화할 때에도 안전성을 높이기 위해 해시함수 등을 이용하여 메시지를 RSA 법과 같아지도록 확장한 후에 암호화를 한다. RSA-OAEP가 대표적인 방식으로 이 기법 역시 결정적 RSA 방식을 확률적 방식으로 바꾸어 준다.

파일과 그것의 해시값을 보관하여 파일의 무결성 서비스를 제공할 수 있지만 공격자에게 쉽게 공격당할 수 있으므로 암호학적으로 안전한 무결성 서비스를 제공하기 위해서는 해시함수 대신에 보통 MAC 함수를 사용한다. 무결성 서비스와 관련하여 13장에서 프로토콜을 통해 아웃소싱된 데이터의 무결성을 확인하는 방법을 살펴본다. 비트 약속이란 4장에서 살펴본 키 제어 요구사항을 위해 사용한 것처럼 어떤 값을 숨긴 채 제시하기 위해 사용할 수 있다. 숨긴 상태로 제시하지만 제시한 사용자가 나중에 값을 공개할 때 원래 제시된 값 대신에 다른 값을 제시할 수 없어야 한다. 따라서 비트 약속에서 사용하는 해시함수는 일방향성은 물론 충돌회피성도 제공해야 한다.

다음 절부터는 지금까지 살펴본 것들을 제외한 해시함수를 활용한 응용을 자세히 살펴본다.

## 2. 패스워드

리눅스와 같은 다중 사용자 운영체제는 물론 각종 웹 서비스에서 사용자를 인증할 때 가장 널리 사용하는 기법이 패스워드 인증 기법이다. 이 기법은 사용자가 입력한 패스워드와 저장된 패스워드를 비교하여 사용자를 인증한다. 패스워드는 장기간 키를 안전하게 유지하기 위해서도 널리 사용하고 있다. 보통 장기간 키는 패스워드로부터 생성한

대칭키로 암호화하여 유지한다. 예를 들어 공동인증서에서 개인키를 이와 같은 방식으로 보호하고 있다.

패스워드의 가장 큰 문제는 쉽게 예측이 가능하다는 것이다. 사용자들은 7, 8자의 짧은 패스워드를 기억하는 것조차 어려워하는 것으로 조사되어 있다. 7, 8자의 영문자와 숫자만으로 구성된 패스워드는 컴퓨터를 사용하면 쉽게 전사 공격하여 알아낼 수 있다. 따라서 최근 웹 서비스는 특수문자, 숫자, 대소문자를 조합하도록 요구하고 있으며, 길이도 일정 길이 이상이 되도록 요구하는 경우도 많다. 또한 패스워드 추측 공격을 방어하기 위해 시도할 수 있는 횟수를 보통 제한하고 있다. 그러나 사용자의 패스워드를 알아낼 방법은 다양하며, 경로도 다양하다.

## 2.1 패스워드 해싱

패스워드 해싱은 등록된 패스워드를 보호하기 위한 기술이다. 사용자의 패스워드를 보호하지 않은 상태로 시스템에 유지하면 시스템이 해킹되었을 때 노출될 수 있다. 이와 같은 허점 때문에 패스워드 대신에 패스워드의 해시값을 유지하며, 이를 패스워드 해싱이라 한다. 패스워드 해싱을 사용하면 패스워드 해시값이 저장된 파일이나 데이터베이스가 노출되어도 해시함수의 일방향성 때문에 공격자가 패스워드를 얻어내는 것이 어렵게 된다. 이 방식에서 사용자가 패스워드를 입력하면 그것을 직접 확인할 수 없고, 그것의 해시값을 계산한 후에 저장된 해시값과 비교하여 확인한다.

패스워드 해싱을 사용하더라도 사용자는 추측하기 쉬운 패스워드를 사용하는 경향이 많으며, 어려운 패스워드를 사용하더라도 사전 공격(dictionary attack)과 같은 컴퓨터를 이용한 전수 조사방식을 사용하면 패스워드를 알아낼 확률이 높다. 사전공격에 대한 취약성을 극복하기 위해 보통 두 가지 추가적인 기법을 사용한다. 하나는 패스워드를 해싱할 때 패스워드만 해시하지 않고 **소금**(salt)이라고 하는 랜덤값과 함께 해시한다. 다른 하나는 일반 해시함수 대신에 계산 속도가 느린 해시함수를 사용한다.

소금의 효과를 높이기 위해서는 사용자마다 다른 랜덤 값을 사용해야 한다. 이 경우 사용자가 입력한 패스워드를 확인하기 위해서는 입력한 패스워드와 소금을 결합하여 해시값을 계산해야 한다. 따라서 서버가 해당 사용자의 소금값을 알기 위해서는 이를 보관할 수밖에 없으며, 보통 해시값과 함께 유지한다. 이 때문에 기대한 것만큼의 사전 공격 비용을 높여주는 효과를 얻기 어렵다.

그러면 실제 소금의 사용이 사전 공격에 어떤 효과가 있는지 살펴보자. 먼저 소금을 사용하지 않는 시스템에서 패스워드를 보관한 파일을 공격자가 확보하였다고 하자. 공격자는 $n$개의 단어에 대해 해시값을 계산한 후에 확보한 파일 내부에 있는 해시값과 비교하여 공격하게 된다. 소금을 사용하는 시스템에서 공격자가 $m$개의 사용자 정보가 있는 파일을 확보하였다고 하자. $m$개의 사용자 정보가 있으므로 소금값도 $m$개가 존재할 것이다. 따라서 이전과 동일한 공격을 하기 위해서는 각 단어마다 $m$개의 해시값이 필요하며, 총 $mn$개의 해시값이 필요하다. 따라서 공격 비용이 증가하였지만, 계산적으로 어려울 정도로 증가된 것은 아니다. 하지만 소금값의 사용은 사전 공격에 대한 공격 비용을 높이는 효과뿐만 아니라 같은 패스워드를 가진 사용자의 해시값이 다르게 되어 해시값만 보면 두 사용자가 같은 패스워드를 쓰고 있는지 알 수 없게 되는 이점도 있다.

Provos와 Mazieres[1]는 저장된 패스워드의 안전성을 높이기 위해 Bcrypt라는 패스워드 해시함수를 개발하였다. 이 함수는 다른 암호알고리즘과 달리 알고리즘의 수행 속도를 느리게 만들 수 있다. 보통 이와 같은 함수는 반복횟수를 통해 해시값 계산에 소요되는 시간을 조절한다. 반복횟수가 $c$이면 해시함수를 $c$번 연속적으로 적용하여 최종값을 계산한다. 수행 속도가 느려지면 그만큼 공격자 입장에서는 사전 공격하는 비용이 증가하게 된다. 하지만 수행 속도를 너무 느리게 하여 정상 서비스 이용까지 불편하게 만들 수는 없다.

2013년에 패스워드 해싱의 안전성을 높이기 위해 NIST가 표준을 제정할 때 사용한 방식처럼 공개 대회를 개최하였으며, 대부분의 참가 알고리즘은 이 장 4절에서 설명하는 키 유도 함수 형태의 함수가 제출되었다. 이 대회는 2015년에 종료되었으며, 우승자는 룩셈부르크 대학의 Biryukov 등이 제안한 Argon2이다[2].

## 2.2 가장 취약 요소 원리

**가장 취약 요소 원리**(weakest link principle)란 전체 시스템에서 가장 취약한 부분이 시스템의 안전성을 좌우한다는 것을 말한다. 패스워드 인증에 가장 취약 요소 원리를 적용하여 보자. 공격자는 크게 클라이언트, 네트워크, 서버 3가지 경로를 통해 패스워드 인증시스템을 공격할 수 있다. 예를 들어 공격자는 클라이언트 단말을 공격하여 키보드 스니핑(sniffing)과 같은 악성 소프트웨어를 불법적으로 설치한 다음 사용자의 패스워드를 알아낼 수 있다. 네트워크의 경우에는 프로토콜의 취약점을 활용하여 공격할 수 있고, 서버에 침입하여 불법적으로 관리자 권한을 얻어 패스워드가 저장된 파일을 확보할 수 있다. 따라서 클라이언트, 네트워크, 서버 모두 동일 수준의 안전성을 제공하지 못하면 가장 수준이 낮은 것이 공격의 주된 목표가 되며, 전체 시스템의 안전성은 해당 수준에 의해 결정된다.

## 2.3 키 관리와 하드웨어

장기간 키는 보통 패스워드 기반 키로 암호화하여 비휘발성 메모리에 유지한다. 따라서 보관된 파일이나 데이터베이스에서 암호문을 탈취하여 패스워드 추측 공격을 할 수 있다. 이 때문에 장기간 키가 저장된 파일이나 데이터베이스에 대한 접근이 어려워야 한다. 예를 들어 처음에는 암호화된 공동인증서의 개인키를 하드디스크에 저장하여 사용하였으며, 저장된 위치를 누구나 쉽게 찾을 수 있었다. 이 때문에 사용자가 자유롭게 이동, 복사, 백업할 수 있는 이점이 있었지만, 공격자에게 쉽게 노출될 수 있어 보안 측면에서는 바람직한 키 관리 방법이 아니었다.

이 문제를 해결하는 방법은 크게 두 가지 방법이 있다. 하나는 화이트박스 기술을 이용하는 것이고, 다른 하나는 보안 하드웨어를 이용하는 것이다. 화이트 박스 기술은 사용하는 암호키가 고정되어 있을 때 사용할 수 있는 기술이다. 원래 암호화 함수는 키와 메시지를 입력으로 받는데, 사용하는 키가 항상 고정되어 있으면 이 키를 고정적으로 사용하는 함수로 암호화 함수를 바꿀 수 있다. 이렇게 바뀐 함수의 코드를 난독화(obfuscation)하면 소프트웨어 실행 파일이나 소프트웨어 실행되는 동안 메모리를 관찰하여도 사용하는 암호키를 얻을 수 없다[3]. 현재 KB국민은행 사설 인증서가 이 기술을 사용하고 있다.

이동 통신사는 각 가입자를 식별하고 과금을 정확하게 하기 위해 USIM을 사용하고 있으며, 이 USIM에 기기를 인증할 때 사용하는 대칭키를 유지한다. 또 초기 IPTV 셋톱박스나 이전 스카이라이프는 스마트카드에 수신제한시스템에 필요한 키를 저장하였다. USIM이나 스마트카드는 모두 보안 하드웨어의 한 종류이다.

참고로 스마트폰을 이용하여 인터넷 뱅킹하는 것이 보편화됨에 따라 공동인증서를 USB 대신에 휴대전화에 저장하여 사용하는 형태로 바뀌었고, 이것의 안전성을 높이고자 인증서를 핸드폰 USIM에 저장하여 주는 서비스도 등장하였다. 인증서를 USIM에 저장하고 USIM에 PIN 번호를 설정해 놓으면 현재 사용가능한 방법 중 가장 안전하게 개인키의 노출을 막을 수 있다. 더욱이 USIM은 연산 수행 능력이 있어 키를 USIM 밖으로 이동하여 암호화 연산을 수행하지 않고 입력 데이터를 받아 USIM 내에서 연산을 수행한 후에 결과만 돌려줄 수 있다.

USIM과 같은 보안 하드웨어를 이용하는 것을 좀 더 구체적으로 살펴보자. 장기간 암호키를 가장 안전하게 보관하는 방법은 보안 하드웨어를 이용하는 것이다. 보안 하드웨어는 기본적으로 2-factor 인증을 제공한다. 그 사용자의 하드웨어 기기와 그 기기를 구동하기 위한 패스워드가 필요하다. 보안 하드웨어는 자체 암호 모듈을 탑재하고 있어 키를 외부로 내보내지 않고 키를 이용한 모든 연산을 자체적으로 수행할 수 있다. 이와 같은 하드웨어를 HSM(Hardware Security Module)이라 한다.

보안 하드웨어는 조작 불가능한(tamper-proof) 기능을 탑재할 수 있다. 보통 조작을 탐지할 수 있고, 조작된 것이 탐지되면 암호키를 포함하여 메모리에 있는 모든 데이터를 자동 삭제한다. 이 때문에 공격자가 보안 하드웨어를 확보하더라도 그 하드웨어에 저장된 키를 알아내는 것이 매우 어렵다. 따라서 이와 같이 하드웨어를 사용할 경우 키의 용도에 따라 별도 백업을 하는 것이 필요하다.

보통 보안 하드웨어를 사용하면 추가 비용이 발생하고 사용 편리성이 저하될 수 있지만 지금은 기존 USB 드라이브와 유사한 형태의 키 관리를 위한 보안 하드웨어가 저렴하게 판매되고 있다. 더욱이 대부분의 사용자가 보유하고

Msg 1. $A \rightarrow S: \quad A$
Msg 2. $S \rightarrow A: \quad N_S, \{K\}.K_\pi$
Msg 3. $A \rightarrow S: \quad \{N_S\}.K$

<그림 11.1> 패스워드 기반 키 확립 프로토콜

있는 모바일폰에는 USIM이라고 하는 간단한 형태의 보안 하드웨어가 포함되어 있어, 이에 대한 활용도가 높아지고 있다.

## 2.4 패스워드 기반 암호프로토콜

랜덤하게 생성한 대칭키는 사용자가 기억하기 어렵기 때문에 패스워드로부터 대칭키를 생성하여 사용하는 경우가 있다. 이와 같이 생성하면 키를 비휘발성 메모리를 유지하지 않고 필요할 때 생성하여 사용할 수 있다. 하지만 이와 같은 키는 패스워드 추측 공격에 취약하기 때문에 보통 로컬에서만 사용한다. 예를 들어 이미 언급한 바와 같이 공동인증서의 개인키는 사용자 패스워드로부터 유도된 대칭키로 암호화하여 유지한다.

우리가 무선랜을 사용할 때 패스워드를 입력하는 경우가 많다. 무선 장치와 무선 AP는 이 패스워드로부터 대칭키를 생성하여 상호 인증한다. 이 경우에는 패스워드로부터 생성된 대칭키를 로컬에서만 사용하는 것이 아니라 무선 장치와 무선 AP 간의 메시지를 교환할 때 사용하고 있다. 이 때문에 이 프로토콜은 근본적으로 패스워드 추측 공격에 취약하다. 이에 WPA3부터는 이 절에서 소개하는 패스워드 기반 프로토콜을 사용하여 안전하지 못한 패스워드를 선택하더라도 더 강건하게 무선 장치와 무선 AP가 상호인증하고 암호 채널을 구축할 수 있도록 해준다.

패스워드로부터 대칭키를 생성하는 방법은 이 장 4절에서 설명하는 키 유도 함수를 이용하며, 이 함수는 내부적으로 해시함수나 MAC을 이용한다. 앞서 언급한 바와 같이 패스워드로부터 유도된 대칭키는 패스워드 추측 공격에 취약할 수 있어 로컬에서만 주로 사용하지만, Bellovin과 Merrit가 제안한 EKE(Encrypted Key Exchange) 프로토콜[4, 5]을 사용하면 패스워드를 이용하더라도 패스워드 추측 공격에 강건한 키 확립 프로토콜을 만들 수 있다. 이 프로토콜의 기본적인 원리는 패스워드로부터 생성된 대칭키는 랜덤한 값만 암호화하여 패스워드 추측 공격을 못하게 만드는 것이다. 즉, 명백한 잉여정보는 이와 같은 키로 절대 암호화하면 안 된다. 하지만 랜덤한 값만 암호화한다고 패스워드 추측에 공격에 강건한 것은 아니다. 암호화된 랜덤 값을 나중에 어떻게 사용하는지에 따라 여전히 패스워드 추측 공격에 취약할 수 있다. 또한 암호화한 랜덤값이 어떤 특징을 가지고 있으면 이 역시 패스워드 추측 공격에 도움을 줄 수 있다.

그림 11.1의 프로토콜은 패스워드 $\pi$로부터 생성한 대칭키 $K_\pi$를 이용하여 랜덤 대칭키 $K$를 암호화하고 있다. 따라서 랜덤한 값을 암호화하고 있기 때문에 패스워드 추측 공격에 강건한 프로토콜이라고 생각할 수 있다. 실제 공격자는 패스워드를 추측하여 메시지 2의 암호문을 복호화하더라도 성공여부를 확인할 수 없다. 그러나 메시지 3에서 세션키를 이용하여 $N_S$를 암호화하였으므로 공격자는 메시지 2와 3에 있는 암호문을 함께 이용하면 패스워드 추측 공격을 할 수 있다. 공격자는 추측한 패스워드로 메시지 2를 복호화한 다음 결괏값을 이용하여 메시지 3의 암호문을 복호화하고, 얻은 값이 메시지 2에 포함된 난스 값과 같은지 비교함으로써 자신의 추측 여부가 맞는지 확인할 수 있다.

그림 11.1의 프로토콜 메시지 2에서 랜덤한 RSA 공개키 $(n, e)$를 $K$ 대신 암호화하여 전달하면 $A$는 이 공개키로 $N_S$를 암호화하여 되돌려주는 형태로 프로토콜을 수정하였다고 하자. 이 경우 메시지 2는 랜덤 값을 암호화하고 있으며, 이 값으로부터 대응되는 개인키를 공격자는 생성할 수 없으므로 메시지 3의 암호문을 복호화할 수 없다. 따라서 패스워드 추측 공격에 강건한 프로토콜이라고 생각할 수 있다. 하지만 RSA 공개키에서 사용하는 $n$과 $e$는 특수한 성질을 가진 수이다. 특히, $n$은 홀수이어야 하고, 소인수분해가 어려워야 하며, 일정한 크기의 수이어야 한다. 따라서 공격자는 메시지 2의 암호문에 대한 패스워드 추측 공격을 하면 자신의 추측이 올바르다는 것은 확신할

수 없지만 어떤 경우에는 확실히 틀렸다는 것을 알 수 있다. 이를 이용하면 여러 패스워드를 후보에서 배제할 수 있다. 공격자는 패스워드 추측 공격을 통해 패스워드의 범위를 축소할 수 있기 때문에 이 프로토콜은 패스워드 추측 공격에 취약한 프로토콜이 된다. 패스워드를 추측하여 공격을 시도하였을 때 시도한 패스워드가 올바르다고 알 수 있는 경우는 당연히 취약한 프로토콜이지만 틀렸다고 알 수 있으면 후보를 배제할 수 있기 때문에 이 역시 취약한 프로토콜이 된다. 이 프로토콜을 개선하기 위해 $e$만 암호화하는 방법을 생각할 수 있는데, 이 경우에 $e$의 특성을 이용하여 유사한 공격을 할 수 없도록 추가적인 조치가 필요하다.

Bellovin과 Merrit가 제안한 EKE는 Diffie-Hellman 키 동의 프로토콜에서 교환하는 $g^a$와 $g^b$를 패스워드 기반 키로 암호화하고 있다. 두 값은 랜덤값이며, 추측한 패스워드로부터 얻은 두 값으로부터 $g^{ab}$를 계산할 수 없기 때문에 패스워드 추측 공격이 가능하지 않다. 이 경우에도 사용하는 군에 따라 추측한 패스워드를 이용하여 얻은 값이 해당 군의 요소인지 여부를 확인하여 패스워드를 배제할 수 있으면 여전히 패스워드 추측 공격에 취약한 프로토콜이 된다. 또 지금까지 소개한 패스워드 기반 프로토콜들은 두 사용자가 패스워드를 공유하고 있다고 가정하고 있지만, 대부분의 패스워드를 사용하는 클라이언트-서버 기반 서비스에서 서버는 실제 패스워드 대신에 패스워드의 해시 값을 가지고 있다. 따라서 이와 같은 환경에서는 제시된 패스워드 기반 프로토콜을 그대로 적용할 수 없다. 참고로 앞서 소개한 무선 장치와 무선 AP는 장치와 AP가 패스워드를 공유하고 있는 형태이다.

EKE는 패스워드를 공유한 클라이언트와 서버가 패스워드 기반 대칭키를 이용하여 상호 인증하고 세션키를 확립하게 해주는 프로토콜이다. 최근에 패스워드를 공유하지 않은 상태에서 패스워드를 이용하여 상호 인증하고 세션키를 확립하게 해주는 프로토콜도 제안되고 있다. 이와 같은 프로토콜을 aPAKE(Asymmetric Password Authenticated Key Exchange)이라 한다.

사용자와 서버가 서로 인증된 공개키 쌍이 있으면 이를 이용하여 상호 인증하고 필요한 세션키를 확립하여 비밀 통신을 할 수 있다. 하지만 사용자가 한 기기만 사용하는 것이 아니라 여러 장치를 이용하여 서비스를 받아야 할 경우에는 공개키 쌍을 모든 기기에 복사하여 유지해야 한다. 이 문제를 개선하기 위해 공개키는 서버에 등록하고 개인키는 패스워드로부터 생성하는 방법을 생각해 볼 수 있다. 하지만 이 경우 패스워드 추측 공격이 가능하다. 추측 공격을 어렵게 하기 위해 소금의 사용을 생각해 볼 수 있지만 그보다는 개인키를 서버에 암호화하여 유지하고 오직 패스워드를 알고 있는 사용자만 그 암호문을 복호화할 수 있도록 하는 방법을 생각해 볼 수 있다. 즉, 필요할 때마다 올바른 패스워드를 제시하면 암호화된 자신의 개인키를 서버로부터 얻어 복호화하여 얻을 수 있다.

이 방법은 FIDO가 사용하는 방법과 유사한 측면이 있다. FIDO는 공개키는 서버에 등록하고 클라이언트 소프트웨어는 개인키를 유지하는 방법을 사용한다. 반면에 aPAKE는 개인키를 암호화된 상태로 서버에 유지한다. 하지만 패스워드 기반 대칭키로 암호화하여 유지하는 것은 아니며, 사용자는 서버로부터 인증할 때 암호화된 개인키를 받지만 이 암호문을 복호화하기 위한 키는 서버와 패스워드 기반 프로토콜을 수행하여 얻는다.

$$
\begin{array}{ccc}
r \in_R \mathbb{Z}_q^*, h = H_q(x) & & \\
\alpha = hg^r & \xrightarrow{\text{①} \quad \alpha} & k \in_R \mathbb{Z}_q^* \\
\gamma = \beta v^{-r} = h^k g^{kr} g^{-kr} = h^k & \xleftarrow{v, \beta \quad \text{②}} & v = g^k, \beta = \alpha^k \\
R = H(x||v||\gamma) & &
\end{array}
$$

<그림 11.2> OPRF 프로토콜

OPAQUE은 aPAKE 프로토콜 중 하나이며, 현재 인터넷 표준으로 준비되고 있다[6]. OPAQUE는 내부적으로 OPRF(Oblivious Pseudo Random Function)라는 것을 이용한다. OPRF는 상호작용 프로토콜이며, 그림 11.2와 같이 진행된다. 여기서 $g$는 위수가 소수 $q$인 군의 생성자이다. 최종적으로 사용자는 $R$을 얻게 되는데, 서버는 사용자의 입력 $x$를 알 수 없으며, 사용자는 서버의 입력 $k$를 알 수 없다. 사용자 $A$의 패스워드가 $\pi_A$일 때 OPAQUE 프로토콜에서 사용자 등록 과정은 다음과 같이 진행된다.

- 단계 1. 서버와 사용자 $A$는 각각 $\pi_A$와 $k_A$를 입력으로 사용하여 OPRF 프로토콜을 진행한다. $A$는 OPRF 수행 결과로 얻은 $R = H(\pi_A||g^{k_A}||H_q(\pi_A)^{k_A})$를 KDF에 입력하여 인증 암호화 필요한 대칭키를 계산한다.

이 키를 이용하여 $+K_A$, $-K_A$, $+K_S$를 인증 암호화한다. 그 결과를 $\text{Env}_A$라 하자.

- 단계 2. $A$는 $+K_A$, $\text{Env}_A$를 서버에 전달하면, 서버는 수신한 값과 $+K_S$, $-K_S$, $g^{k_A}$, $k_A$를 사용자 $A$의 등록값으로 저장한다. 사용자마다 다른 서버 공개키 쌍을 사용하지 않으면 이들을 사용자 등록값으로 저장할 필요는 없다.

OPRF의 특성 때문에 사용자 패스워드는 서버를 포함하여 노출되지 않는다.

등록된 사용자는 다음을 진행하여 상호 인증과 세션키 확립을 한다.

- 단계 1. 사용자 $A$가 접속을 시도하면 서버는 $\text{Env}_A$를 전달한다.
- 단계 2. 서버와 사용자 $A$는 각각 $\pi_A$와 $k_A$를 입력으로 사용하여 OPRF 프로토콜을 진행한다. 사용자 $A$는 이를 통해 등록 과정 때 얻은 동일 $R$을 얻게 되며, 이를 이용하여 단계 1에서 수신한 $\text{Env}_A$를 복호화하여 자신의 공개키 쌍을 얻게 된다.
- 단계 3. 서버와 $A$는 자신들의 공개키 쌍을 이용하여 키 확립 프로토콜을 진행한다.

$\text{Env}_A$를 복호화하기 위해서는 등록 과정과 동일한 OPRF를 수행해야 하며, 이를 위해서는 등록 과정에 사용한 사용자 패스워드를 알아야 한다.

## 3. 일회용 패스워드

**OTP(One-Time Password)**는 기존 계정명/패스워드 방식보다 안전한 인증 메커니즘을 제공하기 위해 매번 새로운 패스워드를 사용하는 방식을 말한다. 이 기법은 매번 패스워드를 변경하면 패스워드가 노출되어도 안전성에 문제가 없다는 것에 착안한 기법이다. 이 기법을 사용하기 위해서는 사용자와 사용자 패스워드를 확인하는 서버가 매번 같은 패스워드를 생성할 수 있어야 한다. 이를 위해 보통 사용자와 서버 간의 대칭키를 공유하고 있으며, 이 키를 이용하여 패스워드를 생성한다.

클라이언트와 서버가 대칭키를 이용하여 매번 다르지만 동일한 패스워드를 생성하기 위해서는 공유한 대칭키 외에 입력으로 사용할 다른 값이 필요하다. 이를 위해 클라이언트와 서버가 서로 약속한 값을 이용할 수 있고, 한 쪽이 생성한 값을 전달받아 사용할 수 있다. 전자는 클라이언트와 서버가 약속이 어긋나지 않게 하는 것이 필요하므로 동기화 방식이라 하고, 후자는 비동기화 방식이라 한다. 보통 동기화 방식에서는 시간 또는 카운터를 이용한다. 꼭 대칭키를 공유하고 이를 이용하여 일회용 패스워드를 사용할 필요는 없다. 이 장 6 절에서 소개하지만 해시체인과 같은 기술을 사용하여 일회용 패스워드를 생성할 수 있다.

OTP를 가장 먼저 활용한 응용은 인터넷 뱅킹이다. 인터넷 뱅킹은 하드웨어 기반 OTP를 통해 보안 강도를 높이고자 하였다. 인터넷 뱅킹에서 하드웨어 기반 OTP를 사용한 이유는 그 당시 사용자들이 주로 PC를 이용하여 인터넷 뱅킹을 사용하였기 때문이다. 특히, 인터넷에 연결된 어느 컴퓨터에서도 쉽게 인터넷 뱅킹을 사용할 수 있어야 했는데, 소프트웨어 기반 OTP를 사용하면 이것이 가능하지 않았다. 소프트웨어 기반 OTP를 사용하기 위해서는 인터넷 뱅킹을 사용할 컴퓨터마다 추가적인 소프트웨어 설치가 필요하다. 더 큰 문제는 뱅킹 서버와 공유한 대칭키를 해당 소프트웨어가 접근할 수 있도록 해야 하는 문제점이 있다. 이와 달리 하드웨어 기반 OTP는 기기 화면에 등장하는 6자리 숫자만 화면에 입력하면 되기 때문에 다른 컴퓨터를 사용하더라도 OTP 때문에 번거로워지는 것이 전혀 없다.

하드웨어 기반 OTP는 기기 화면에 나타난 숫자만 입력하는 방식이기 때문에 동기화 방식으로 동작할 수밖에 없다. 기기 내부에 서버와 공유한 대칭키를 유지하며, 이 키와 시간이나 카운터를 이용하여 값을 계산한 다음 이 값으로부터 화면에 나타날 숫자를 만들어 표시하는 방식으로 동작한다. 서버는 각 사용자에게 발급된 기기를

알고 있으며, 각 기기에 저장된 대칭키를 알고 있다. 따라서 서버는 기기와 동일한 방법을 통해 숫자를 계산하여 비교함으로써 사용자를 인증한다.

하드웨어 기반 OTP의 요구사항은 다음과 같다.

- R1. [불위조성] 패스워드는 그 기기와 서버 외에는 생성하는 것이 계산적으로 어려워야 한다.
- R2. [독립성] 이전에 사용된 패스워드로부터 현재 사용할 패스워드를 제3자가 계산하는 것이 계산적으로 어려워야 한다.
- R3. [간결성] 제시된 값은 사용자가 쉽게 읽을 수 있어야 하며, 사용자가 화면에 쉽게 입력할 수 있어야 한다.
- R4. [경제성] 하드웨어적으로 구현이 경제적이어야 한다.

R1은 생성된 패스워드로부터 해당 패스워드를 생성하기 위해 사용된 대칭키를 구하는 것이 계산적으로 어려워야 한다. 보통 기기에 저장되어 있는 대칭키를 MAC 키로 사용하여 현재 시각 또는 카운터 값에 대한 MAC값을 계산하여 패스워드를 생성한다. 따라서 MAC 값의 안전성 때문에 이 요구사항은 쉽게 충족된다. 더욱이 MAC 값 자체를 사용하는 것이 아니라 MAC 값으로부터 6자리 숫자를 만들어 패스워드로 사용하기 때문에 R1에서 사용된 패스워드로부터 키 정보가 노출되는 것은 가능하지 않다고 생각해도 된다.

각 패스워드를 매번 독립적으로 생성하면 R2도 충족할수 있다. 사용자가 해시값 자체를 입력해야 하면 값 자체가 길어 번거로우며, 잘못 입력하는 경우도 많아진다. 따라서 계산된 값으로부터 적당히 짧은 길이의 값을 계산하여 보여주어야 한다. 보통 여섯 자리 정도의 정수로 바꾸어 사용한다. 여섯 자리 정수를 사용하면 경우의 수가 비교적 적어 전사공격이 가능할 수 있다. 하지만 매번 이 값이 바뀌기 때문에 큰 문제가 되지는 않는다. 물론 안전성을 높이기 위해 입력 오류 횟수 제한, 암호 채널 사용 등 추가 보안 조치를 취할 필요는 있다. 많은 사용자들이 OTP를 사용하도록 유도하기 위해서는 무료로 보급하거나 비싸지 않아야 한다.

## 3.1 OTP의 종류

OTP는 크게 클라이언트와 서버 간의 동기화가 필요한 방식과 필요하지 않은 방식으로 구분할 수 있다. 이 절에서는 클라이언트와 서버가 대칭키를 공유하고, 이 대칭키를 이용하여 일회용 패스워드를 생성하는 방식의 OTP만 고려한다. 대칭키를 활용하는 비동기화 방식은 한 쪽이 랜덤한 값을 생성하여 다른 쪽에 주어 이 값을 이용하여 패스워드를 생성한다. 이와 같이 생성하기 때문에 이 방식을 시도응답 방식이라 하며, 5장에서 살펴본 난스 기법과 차이가 없다. 서버가 $N_S$를 전달하면 클라이언트는 공유한 대칭키로 이 값을 암호화하거나 공유한 대칭키를 MAC 키로 사용하여 서버가 준 난스에 대한 MAC 값을 계산하여 회신하는 방식이다. 이 방식에서 서버가 주는 난스값은 매번 반드시 달라야 한다. 서버에 사용자 공개키를 유지하면 MAC 대신에 전자서명을 이용할 수도 있다.

이와 같은 방식을 사용하기 때문에 비동기화 방식은 별도 하드웨어 장치를 이용하여 처리하는 것이 어렵다. 서버가 전달한 난스값을 이 장치에 전달해야 하는데, 컴퓨터와 장치 간 통신이 가능하지 않으면 사용자가 직접 입력해 주어야 한다. 하지만 이것은 번거로울 뿐만 아니라 기기가 키보드와 같은 입력 수단을 제공해야 하는 문제점도 있다.

또 비동기 방식은 짧은 값을 전달하여 인증하는 것이 아니기 때문에 OTP가 아니라 간단한 인증 프로토콜이라고 하는 것이 더 정확할 수 있다. 물론 이 경우에도 MAC 값 전체 대신에 해당 값으로부터 작은 길이의 패스워드를 생성하여 전달할 수 있다. 이것이 오히려 전체 MAC 값이나 암호문이 노출되지 않아 더 안전할 수 있다. 물론 전자서명의 경우에는 작은 길이의 패스워드로 바꾸어 공개키만 가지고 있는 서버가 확인할 수 있도록 하는 것은 어려울 수 있다.

동기화 방식은 크게 시간 동기화 방식과 사건 동기화 방식으로 다시 구분할 수 있다. 시간 동기화 방식은 클라이언트와 서버 간의 공유된 대칭키와 절대 시간을 이용하여 패스워드를 생성한다. 클라이언트와 서버가 같은

<표 11.1> 시간 동기화와 사건 동기화 방식의 비교

| | 시간 동기화 | 사건 동기화 |
|---|---|---|
| 생성 | 정해진 시간 간격마다 자동 생성<br>$f(K,T)$, $T$: 시간 | 필요할 때마다 생성<br>$f(K,C)$, $C$: 카운터 |
| 동기화 | 절대 시간을 이용함<br>시스템 간 클럭동기화 필요 | 공유 카운터를 사용함<br>look-ahead 파라미터 $s$ 사용 |
| 미래값 | 확보할 수 없음 | 계속 생성하여 얻을 수 있음 |
| 편리성 | | 클럭 동기화가 필요 없음 |
| 문제점 | 입력 도중에 패스워드가 바뀔 수 있음 | 각 기기마다 별도 카운터를 유지해야 함 |
| 공격가능성 | 유효기간 내 재사용 가능<br>한번 인증된 값을 재사용할 수 없어야 함 | 미래값 확보 가능<br>패스워드의 유효기간이 별도 없음 |

패스워드를 생성하기 위해 특정 시간 간격으로 패스워드를 생성한다. 예를 들어 1분마다 새로운 패스워드를 생성하는 방식을 사용할 수 있으며, 클라이언트와 서버가 정확하게 시간 동기화가 안 되어 있을 수 있기 때문에 서버는 현재 시각을 기준으로 지금까지 사용하지 않은 전후 몇 분에 대한 패스워드를 생성하여 확인하는 방식을 사용하고 있다. 또한 정해진 시간 간격 사이에는 동일 패스워드가 다시 인증될 수 있기 때문에 한 번 사용한 패스워드를 다시 인증에 사용할 수 없도록 중복 검사를 해야 한다.

사건 동기화 방식은 사용자가 요청할 때마다 일회용 패스워드를 생성하는 방식이며, 이를 위해 클라이언트와 서버가 대칭키 외에 카운터를 공유하고 있다. 클라이언트는 다음 카운터 값을 계산한 다음에 이 값을 이용하여 패스워드를 생성하여 전달하는 방식을 사용하고 있다. 사용자가 사용하는 기기의 버튼을 누르면 다음 패스워드를 생성하기 때문에 사용자는 기기의 버튼을 여러 번 눌렀지만 생성한 값을 서버에 전달하지 않을 수 있다. 이 경우 클라이언트와 서버 간의 카운터 값 차이가 너무 벌어져 인증을 확인하는 데 어려움이 있을 수 있다. 보통 서버는 자신의 카운터 값을 기준으로 정해진 $s$개를 생성하여 생성된 $s$개 내에 일치하면 카운터 값을 동기화하고 인증하는 방식을 사용하고 있다.

두 방식을 비교하면 표 11.1과 같다. 시간 동기화는 절대 시간을 이용하기 때문에 동기화 측면에서 편리하지만 정해진 간격이 끝나지 않으면 다시 인증하기 위해 기다려야 하는 문제점이 있다. 반대로 사건 동기화는 정해진 간격에 따라 자동 생성하는 것이 아니며, 시간 동기화가 필요 없다는 장점이 있지만, 동기화가 어긋날 경우 이를 다시 동기화하기 번거로울 수 있다. 이에 시간과 사건 동기화 방식을 혼합하여 사용하는 혼합 방식도 제안되었다. 혼합 방식에서는 시간 동기화처럼 정해진 간격마다 절대 시간을 이용하여 자동 동기화되지만, 간격 사이에서는 사건 동기화 방식을 사용한다. 따라서 동기화가 어긋나더라도 일정 시간이 지나면 자동 동기화가 된다.

## 3.2 HOTP

HOTP(HMAC based OTP)[7]는 인터넷 표준으로 제정된 사건 동기화 방식의 OTP이다. 클라이언트와 서버는 카운터 $C$와 대칭키 $K$를 공유하고 있으며, $HS = \text{MAC}.K(C)$를 계산한 다음에 $HS$로부터 여섯자리 숫자를 생성하여 일회용 패스워드로 사용한다. SHA-1기반 HMAC을 사용할 경우 $HS$의 출력은 20byte가 되며, 이 중 4byte를 선택하여 이를 32bit 정수로 취급한 다음 해당 값을 1,000,000으로 나머지 연산을 취하여 여섯 자리 숫자를 만든다. HOTP는 $HS$의 마지막 바이트의 하위 4bit 값을 오프셋으로 사용하여 4byte를 선택한다. 예를 들어 하위 4bit 값이 10이면 11번째 바이트부터 4byte를 선택한다.

# 4. 키 유도 함수

해시함수는 키 유도 함수(KDF, Key Derivation Function)를 만들 때도 사용한다. KDF는 키 동의 프로토콜처럼 어떤 공유된 비밀로부터 또는 사용자의 패스워드로부터 사용하고자 하는 대칭 암호알고리즘이나 MAC에 안전하고 적합한 키를 만들 때 사용하는 함수이다. 패스워드 기반 KDF는 PBKDF라 하며, 일반 KDF와 달리 패스워드 해싱처럼 소금과 반복횟수를 사용한다. 여기서 반복횟수는 수행 속도를 조절하는 용도로 사용하며, 이를 통해 사전 공격에 대한 방어력을 높인다.

KDF를 사용하면 필요한 요구조건을 충족하는 비밀키를 생성할 수 있다. 예를 들어 대칭 암호알고리즘에 따라 사용하지 말아야 하는 키(weak key)가 있는데, 이들을 배제할 수 있도록 KDF를 설계할 수 있다. 인터넷 표준에서는 HMAC을 이용한 KDF를 표준으로 제정하고 있다[8]. 이 표준을 HKDF라 하며, 두 단계로 구성된다. 첫 번째 단계는 두 번째 단계에서 사용할 MAC 키를 생성하는 단계이고, 두 번째 단계는 필요한 만큼의 비트를 얻기 위해 확장하는 단계이다.

첫 번째 단계에서는 임의로 생성한 소금값을 MAC 키로 사용하여 비밀키를 생성하기 위한 입력값에 대한 MAC 값을 계산한다. 예를 들어 Diffie-Hellman 키 동의 과정 후 대칭키를 생성할 경우 MAC.salt($g^{ab}$)를 계산한다. 이 값은 두 번째 단계의 MAC 키 $K$로 사용한다. 원격에 있는 두 사용자가 동일한 키 값을 얻기 위해서는 두 사용자가 모두 동일한 소금값을 사용해야 한다. 따라서 소금값이 없을 때는 길이가 MAC 출력과 같은 0비트 문자열을 소금값으로 사용한다.

두 번째 단계는 필요한 길이의 키를 얻기 위해 확장하는 단계로 $T_1$부터 필요한 만큼의 길이를 모두 얻을 때까지 차례로 $T_i$를 다음과 같이 계산한다.

$$T_i = \text{MAC}.K(T_{i-1}||\text{info}||i)$$

여기서 $T_0$는 길이가 MAC 출력과 같은 0비트 문자열이다. SHA-1기반 HMAC을 사용할 경우, 2개의 128bit 대칭키와 2개의 160bit MAC키가 필요하면 $T_4$까지 계산하여 이들을 사용한다.

인터넷 표준 RFC 2898[9]에는 PBKDF1과 PBKDF2 두 개의 버전의 패스워드 기반 키 유도 함수가 정의되어 있다. PBKDF1은 해시함수를 사용하는 버전으로써 $T_1 = H(\pi||\text{salt})$, $T_2 = H(T_1)$, ..., $T_c = H(T_{c-1})$를 계산한 다음, 마지막 $T_c$에서 필요한 만큼의 비트를 대칭키로 사용한다. 여기서 $\pi$는 사용자 패스워드이고 $c$는 반복횟수이다. PBKDF1은 내부적으로 사용하는 해시함수 출력값의 길이보다 짧은 길이의 대칭키만 생성할 수 있다.

PBKDF2는 MAC 함수를 사용하는 버전으로써 패스워드 $\pi$를 MAC키로 사용하여 다음과 같이 반복횟수 $c$만큼 $U_i$를 계산하여 이들을 모두 XOR하여 필요한 수의 $T_i$를 만든다.

$$\begin{aligned} U_1 &= \text{MAC}.\pi(\text{salt}||i) \\ U_2 &= \text{MAC}.\pi(U_1) \\ \vdots &= \vdots \\ U_c &= \text{MAC}.\pi(U_{c-1}) \end{aligned}$$

$$T_i = F(\pi, \text{salt}, c, i) = U_1 \oplus U_2 \oplus \cdots \oplus U_c$$

응용에서 필요한 키는 생성된 $T_i$로부터 차례로 추출하여 사용한다.

응용에서 패스워드로부터 항상 같은 키 값을 생성해야 할 경우에는 매번 같은 소금값을 사용해야 한다. 패스워드 기반 암호알고리즘 표준이 최초로 제정된 2000년도에는 1,000번 반복하는 것을 권장하였지만 현재 NIST는 가급적 시스템 환경이 허용되는 범위 내에서 최소 10,000번 이상하도록 권장하고 있다.

HKDF는 길이가 고정된 해시함수를 내부적으로 사용하며, 고정된 길이의 해시함수를 이용하여 가변 길이의

출력을 얻기 위해 반복적으로 해시함수를 계산해야 한다. SHA-3는 기존 SHA-2와 달리 스펀지 구조를 이용하기 때문에 출력 길이에 대한 제한이 실제 없다. HKDF처럼 주어진 입력으로부터 필요한 길이의 랜덤 비트 문자열을 주는 함수를 XOF라 하는데, 스펀지 구조을 이용하는 해시 함수는 그 자체가 XOF이다. SHA-3를 표준화할 때 NIST는 SHA-3를 이용한 XOF도 표준화하였으며, 이를 SHAKE라 한다. NIST는 SHAKE를 확장한 cSHAKE도 표준화하였으며, cSHAKE는 SHAKE보다 문자열을 하나 더 받아 필요한 크기의 랜덤 비트 문자열을 출력하여 준다. 이 문자열을 통해 같은 입력이더라도 다른 출력을 얻을 수 있다. SHAKE와 cSHAKE는 HKDF 대신 공유 비밀 정보로부터 필요한 갯수의 랜덤 대칭키를 생성할 수 있다.

# 5. 예지력 증명

6장에서 키 동의 프로토콜을 설명할 때 키 제어 요구사항을 위해 해시함수를 사용한 바 있다. 해당 예제에서 사용자는 랜덤하게 선택한 군 원소를 평문으로 전달하지 않고 대신 해시값을 전달하였다. 따라서 이것을 수신한 사용자는 해시함수의 일방향성 때문에 상대방 사용자가 선택한 군 원소를 알 수 없으며, 해시값을 전달한 사용자는 해시함수의 충돌회피성 때문에 원래 선택한 군 원소가 아닌 다른 값을 나중에 공개할 수 없다. 이 기술을 비트 약속이라 한다. 이것은 동전 던지기 예제에서도 사용한 기법이다.

비트 약속을 이용하여 예지력 증명을 할 수 있다. 예를 들어 축구 경기 시작 전에 축구 경기 결과를 알고 있음을 증명하고 싶다고 하자. 이를 위해 자신이 예측한 결과의 해시값 $H$("KOR3:GER0")을 공개하였다고 하자. 앞서 살펴본 것과 동일하게 일방향성과 충돌회피성 때문에 충돌회피 해시함수를 이용하여 예지력 증명에 문제가 없다고 생각할 수 있다. 하지만 경우의 수가 제한적이면 가능한 모든 경우에 대해 해시값을 계산하여 어떤 결과를 예측하였는지 사전에 알 수 있다. 한 가지 예로 2014년 월드컵 결승을 트위터에 예측하여 성공한 계정이 있었는데, 실제로는 가능한 모든 경우를 게시한 후에 정답을 제외하고는 삭제하여 성공한 것이었다.

# 6. 해시체인

Lamport는 1981년에 **해시체인**(hash chain)이라는 해시함수의 일방향성을 이용한 간단한 인증 방식을 제안하였다[16]. 해시체인은 랜덤한 초깃값을 선택한 후에 그 값에 해시함수를 연속적으로 적용하여 생성한다. 만약 길이가 5인 해시체인을 생성하고 싶으면 랜덤한 초깃값 $s$를 선택한 후에 아래와 같이 생성한다.

$$c_5 = H(s), c_4 = H(c_5), c_3 = H(c_4), c_2 = H(c_3), c_1 = H(c_2), c_0 = H(c_1)$$

여기서 $c_5$부터 $c_1$까지가 해시체인의 값이며, $c_0$는 이 체인의 루트(root)라 한다. 해시체인을 사용할 때에는 생성한 방향과 거꾸로 사용하게 된다.

패스워드 인증 대신에 해시체인을 이용하여 사용자를 인증하고자 하면 사용자는 해시체인의 모든 값($c_5$부터 $c_1$)을 유지하고, 서버에 $c_0$를 등록한다. 그다음 $c_1$부터 차례로 사용한다. 예를 들어 사용자가 $c_1$를 전달하면 서버는 이를 해시한 후에 저장된 $c_0$와 비교하여 일치하면 사용자를 인증하게 된다. 인증이 성공적으로 이루어지면 사용자는 해당 체인값을 버리고, 서버는 저장된 체인값을 수신한 값으로 교체한다. 서버를 포함하여 공격자가 $c_i$를 확보하더라도 해시함수의 일방향성 때문에 다음에 사용할 $c_{i+1}$를 계산할 수 없다.

체인값은 한 번만 사용하기 때문에 해시체인은 일회용 패스워드로 활용할 수 있다. 실제 Bellcore[1] 사는 S/KEY라는 해시체인을 이용한 일회용 패스워드를 개발하였으며, 이 인증 방식은 인터넷 표준(RFC 2289)으로도 채택되어 있다[11]. 이 방식은 앞서 살펴본 하드웨어 OTP처럼 체인값으로부터 작은 패스워드를 생성하여 활용할 수는 없다.

[1]Bellcore는 1983년 AT&T 사가 분할될 때 설립된 Bell Communications Research 사의 약칭이다. 1999년에 Telcordia로 기업명이 변경되었으며, 2011년에 Ericsson 사가 인수하여 Ericsson 사 내부로 흡수되었다.

해시체인을 사용할 때 루트값의 인증이 매우 중요하다. 위 예에서 $c_0$를 등록할 때 공격자가 그것 대신에 자신이 만든 루트값을 등록하게 되면 공격자가 해당 사용자로 시스템에 접속할 수 있게 된다. 이 때문에 서버에 등록할 때 사용자는 루트값을 전자서명하여 전달할 수 있다. 서버가 항상 루트값을 이용하여 체인값을 인증하는 것이 아니기 때문에 유지하는 최종 사용된 체인값의 무결성도 보호되어야 한다.

사용자는 해시체인의 모든 값을 유지할 필요는 없다. 길이가 $n$일 때, 최소한 $c_n$과 다음에 사용할 체인값의 색인을 유지해야 한다. 하지만 이 경우 매번 많은 수의 해시함수를 실행해야 한다. 매번 실행해야 하는 해시함수의 수는 유지하는 값의 개수를 늘려 줄일 수 있다. 실행해야 하는 해시함수의 수와 유지해야 하는 체인값의 개수는 해시체인의 길이에 영향을 받게 된다. 해시체인은 생성한 후에는 길이가 고정되기 때문에 활용하는 응용에 따라 길이와 유지하는 값의 개수를 적절하게 결정해야 한다. 또한 체인에 있는 모든 값을 다 사용하면 해시체인을 새롭게 생성해야 하며, 루트값의 등록절차도 다시 이루어져야 한다.

## 7. 해시 퍼즐

서비스 거부 공격을 방어하기 위해 접속하는 클라이언트를 인증하는 방법이 있다. 이는 불법적인 접근을 차단하여 서버의 불필요한 자원의 낭비를 막고자 하는 것이다. 하지만 인증하는 비용이 높을 경우 오히려 인증 비용 때문에 서비스 거부 공격이 쉬워질 수 있다. 따라서 프로토콜이 진행됨에 따라 점진적으로 인증 비용을 높이는 방법이 있다. 이때 사용할 수 있는 인증 방식 중 하나가 클라이언트 퍼즐이다. 퍼즐의 해결 비용은 정상적인 클라이언트 입장에서는 크지 않지만 서비스 거부 공격을 위해 동시에 여러 클라이언트를 실행해야 하면 부담이 될 정도로 커야 한다. 하지만 오늘날 서비스 거부 공격은 하나의 공격 컴퓨터를 이용하여 공격하는 것이 아니기 때문에 큰 효과는 없을 수 있다.

클라이언트 퍼즐 중 해시함수를 이용하는 퍼즐을 해시 퍼즐이라 하며[12], 해시값이 주어졌을 때 해시의 입력을 찾아내는 것이다. 하지만 일방향 해시함수를 사용하기 때문에 해시값이 주어졌을 때 입력 전체를 찾는 것은 퍼즐로 사용할 수 없다. 따라서 입력의 일부 $k$ 비트를 제외한 나머지 값을 공개하여 공개하지 않은 $k$비트를 찾는 문제를 클라이언트 퍼즐로 사용한다. 이 퍼즐은 $k$의 길이를 조절하여 퍼즐의 답을 찾는 데 필요한 시간을 조절할 수 있다.

특정 메시지에 대한 해시값은 한 번의 해시 연산으로 바로 계산할 수 있다. 비트코인에서는 특정 메시지에 대한 해시값을 계산하지만 그것을 계산할 때 일정한 시간이 소요되도록 만드는 것이 필요하였다. 패스워드 해싱처럼 특정한 횟수를 반복하도록 하여 일정한 시간이 필요하도록 만들 수 있지만 이 경우에는 이 값을 확인하는데 걸리는 시간도 함께 길어지는 문제가 있다. 비트코인에서는 해시값을 계산하는데 일정 시간이 소요되지만 확인은 매우 빠르게 하는 것이 필요하였다. 또 비트코인에서는 값을 계산하는데 소요되는 시간이 결정적이 아니라 확률적이어야 하기 때문에 특정한 횟수를 반복하는 방식은 사용할 수 없다.

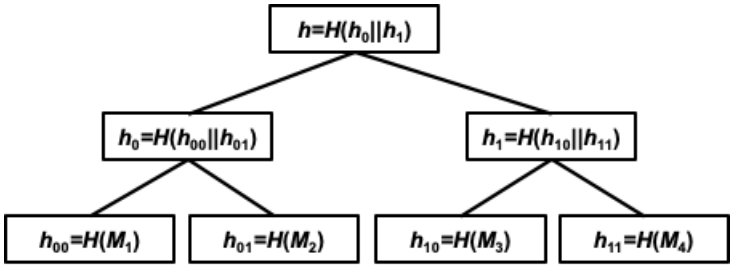
비트코인에서는 이를 위해 특정 입력과 랜덤값을 결합하여 결과 해시값이 특정 수 이하가 되는 랜덤값을 찾는 퍼즐을 사용하였다. 여기서 특정 수 이하란 해시값 전체를 매우 큰 정수로 가정하는 것으로써, 해시값의 최상위 $k$ 비트가 0이 되는 것을 말한다. 이 퍼즐은 어떤 값을 찾는데 최소한 특정 시간이 요구되지만 확인은 한 번의 해시 연산을 통해 확인할 수 비대칭성 특징을 가지고 있는 퍼즐이다. 특정 시간이 요구되기 때문에 이를 **작업 증명**(POW, Proof Of Work)이라 한다.

이 퍼즐은 실제 비트코인에서 가장 먼저 사용한 것은 아니다. 이 기술은 원래 스팸 메일 방지를 위해 제안된 기술이다[13]. 수신자는 메일에 수신자, 메일 내용을 입력으로 사용하는 유효한 해시 퍼즐의 해결책이 포함되어 있어야 정상 메일로 인식한다. 이 때문에 스팸 메일을 보내고자 하는 공격자는 각 수신자마다 새 해시 퍼즐을 해결해야 한다. 따라서 사용하는 작업 증명의 비용을 잘 조절하면 정상 사용자는 비용이 부담되지 않지만 많은 수의 스팸 메일을 전송하고자 하는 공격자는 포기를 유도할 수 있을 정도의 비용이 소요되도록 만들 수 있다.

### 7.1 블록 체인

블록 체인은 현재 비트코인[2]에서 사용하면서 4차 산업혁명의 핵심 IT 기술 중 하나로 주목 받고 있다. 비트코인에서 사용한 블록 체인은 블록 단위로 데이터를 기록하고 이전 블록의 해시값을 포함하여 작업 증명하는 형태로 블록의 해시값을 계산한다. 따라서 이전 블록에 있는 내용을 수정하거나 없는 데이터를 추가하기 위해서는 해당 블록부터 지금까지의 모든 블록을 전부 다시 계산해야 한다. 하지만 작업 증명, 해당 응용에서 사용하는 규칙(비트코인의 경우: 가장 긴 길이의 체인 우선이라는 규칙 사용)과 특성(작업 증명에 참여하는 참여자들과 그들의 생태계) 때문에 이것이 가능하지 않다. 따라서 블록 체인을 첨삭 전용(append-only) 데이터베이스라고도 한다. 블록체인과 비트코인에 대한 자세한 내용은 15장과 16장에서 설명한다.

## 8. 해시 트리




<그림 11.3> 해시트리

문제 해결 방법이 입력의 비례하는 비용이 요구될 때 이 비용을 줄이기 위해 보통 많이 생각하는 방법이 이진 트리를 이용하여 문제를 표현하고 그것의 해결책이 트리 높이에 비례하도록 만드는 것이다. 물론 표현된 이진 트리는 균형 트리이어야 선형 비용을 로그 비용으로 줄일 수 있다.

한 사용자가 $n$개의 값에 전자서명해야 할 때 각 값에 독립적으로 서명하게 되면 총 $n$번의 전자서명이 필요하다. 이 비용은 이진 트리를 이용하여 비용을 줄일 수 있다. 그림 11.3처럼 단말 노드는 각 서명해야 값의 해시값이 되도록 하고 중간 노드의 값은 두 자식 노드의 값을 결합하여 해시한 값이 되도록 이진 트리를 구성한 다음 사용자는 루트값 하나에만 서명을 하면 된다. 특정 값의 전자서명을 누구에게 제시해야 할 경우에는 해당 단말노드와 그 노드부터 루트까지 루트를 제외한 경로에 있는 노드의 형제 노드 값과 루트에 대한 서명값을 제시하면 된다. 이 트리의 높이는 $\log n$에 비례하기 때문에 제시한 값들의 개수도 $\log n$에 비례한다. 이와 같은 서명 방식을 일괄 서명(batch signature)이라 한다[14].

해시 트리는 최초에 Merkle이 제안하였으며[15], 다양한 응용에서 활용하고 있다. 중앙 집중 그룹키 프로토콜에서는 OFT(One-way Function Tree)라는 이름으로 응용되었고, 비트코인의 블록체인에서는 블록에 포함된 트랜잭션들을 해시 트리로 표현한다. 해시 트리는 다른 말로 머클 트리라 한다.

## 9. 해시 기반 서명

보통 공개키 기반 전자서명 기법은 일방향 트랩도어 함수에 의존하며, 이들은 계산적으로 풀기 어려운 수학 문제에 기반하고 있다. 가장 많이 활용하고 있는 인수분해와 이산대수 문제는 양자 컴퓨팅을 이용하면 다차 시간에 해결할 수 있기 때문에 양자 컴퓨팅이 현실화되면 더는 사용하기가 어렵다. 이에 양자 컴퓨팅 환경에서도 안전한 공개키 암호알고리즘에 대한 연구가 활발하게 진행되고 있다. 전자서명의 경우 하나의 대안으로 고려되고 있는 것이 해시

[2]블록 체인은 비트코인에 처음으로 사용된 기술은 아니다.

기반 전자서명이다. 하지만 해시 기반 전자서명은 보통 하나의 개인키로 하나의 메시지만 서명할 수 있는 일회용 전자서명(OTS, One-Time Signature)이다. 이 문제를 극복하고자 해시 기반 일회용 전자서명과 해시 트리를 함께 이용하는 방법이 제안되고 있다.

해시 기반 일회용 전자서명을 최초로 제안한 것은 Lamport이다[16]. Lamport의 1bit 일회용 전자서명에서 공개키는 $H(x)$와 $H(y)$이며, 개인키는 $x$와 $y$이다. 여기서 $x$와 $y$는 매우 큰 정수이다. 이 기법을 이용하여 0을 서명하고 싶으면 $x$를 공개하고, 1을 서명하고 싶으면 $y$를 공개한다. 이 기법을 $n$ 비트 서명으로 확장하고자 하면 $n$개의 $(x, y)$ 쌍이 필요하며, 이들의 해시값이 공개키가 된다. 메시지를 SHA-256를 이용하여 해시한 후에 서명하면 512개의 정수와 512개의 해시값이 필요하다. 즉, 공개키의 길이가 8KB가 된다.

해시 기반 일회용 전자서명에서도 확인키의 인증이 필요하다. 하지만 중앙집중 PKI를 이용하기는 어려울 수 있다. 인증기관이 기존 전자서명을 사용하면 기존처럼 사용할 수 있지만 인증기관도 해시기반 일회용 전자서명을 사용해야 한다면 인증서를 사용하는 것은 간단한 문제가 아니다. 따라서 2장에서 설명한 탈중앙 PKI처럼 각 사용자가 블록체인에 자신의 공개키를 등록하는 방식을 사용해야 할 것으로 보인다.

Wintermitz는 $H(x)$와 $H(y)$를 공개하는 대신에 $x$를 $w$번 해시한 $H^w(x)$를 공개하는 형태로 Lamport의 서명을 개선하였다[17]. $H^w(x)$를 공개키로 이용하면 0부터 $w-1$까지의 값에 서명할 수 있다. $w = 16$이고 $m = 9$를 서명하고 싶으면 $H^9(x)$를 공개하면 된다. 검증자는 받은 값을 $w-9$번 해시하여 공개키와 비교하면 된다. 이 서명의 문제점은 공격자가 서명값을 받아 해시하여 $m$보다 큰 값의 서명을 만들 수 있다. 이것이 가능하지 않도록 추가적인 조치[3]가 필요하다. 메시지를 SHA-256으로 해시하여 서명한다고 생각하고 $w = 256$을 사용하면 공개키는 $H^{256}(x_i)$가 총 32개가 필요하고, $w = 16$를 사용하면 총 64개가 필요하다. 전자의 경우 공개키의 길이는 1KB 밖에 되지 않는다.

해시 기반 일회용 전자서명은 여러 문서로 구성된 해시 트리 루트에 서명함으로써 하나의 문서가 아니라 여러 개의 문서에 서명할 수 있다. 또 다른 방법은 문서 대신에 일회용 공개키를 단말로 하는 해시 트리를 만들고, 이 트리의 루트를 공개키로 사용하면 일회용 서명 방식을 다중 서명 방식으로 확장할 수 있다. 해시 트리의 사용으로 공개키의 길이는 줄였지만 여전히 사용자는 많은 수의 개인키를 유지해야 한다. XMSS[18]는 이 문제를 극복하기 위해 개인키를 독립적으로 생성하지 않고, seed와 트리 위치를 이용하여 의사난수생성기를 이용하여 결정적으로 계산하는 방식을 사용하고 있다. 이 방법을 사용하면 사용자는 seed만 유지하면 된다.

한 해시 트리에 포함할 수 있는 공개키의 수는 제한적이다. 많은 공개키를 포함할수록 트리의 높이는 증가한다. 이 문제는 하나의 해시 트리로 모든 공개키를 표현하지 않고 작은 높이의 여러 개 해시 트리를 이용할 수 있다. 공개키를 유지하는 여러 개의 작은 높이의 해시 트리들의 루트로 구성된 해시 트리의 루트가 최종 공개키가 된다. 이와 같은 트리를 하이퍼트리(hypertree)라 한다.

하지만 여전히 XMSS에서 사용한 공개키를 관리해야 하며, 절대 이전에 사용한 키를 다시 사용하지 않아야 한다. 이 관리를 사용자가 직접 별도로 해야 한다. 이 문제를 개선한 SPHINCS+ 기법(NIST에서 진행 중인 양자내성암호 표준화 작업 3라운드 후보 중 하나)[19]도 있지만 여전히 기존 RSA 서명이나 타원곡선 기반 서명에 비교하면 해시 기반 서명이 효과적이지 못하다는 것은 직관적으로 알 수 있다.

# 참고문헌


[1] Niels Provos, David Mazières, "A Future-Adaptable Password Scheme," Proc. of the USENIX Annual Technical Conf., pp. 32–32, 1999.

[2] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, "Argon2: the Memory-hard Function for Passowrd Hashing and Other Application," `https://www.cryptolux.org/index.php/Argon2`, Dec. 2015.

[3] Brecht Wyseur, "White-box Cryptography: Hiding Keys in Software", MISC magazine, Apr. 2012.


[3] checksum을 계산하고 이 checksum까지 서명한다.

[4] S.M. Bellovin, M. Merritt, “Encrypted Key Exchange: Password-based Protocols Secure Against Dictionary Attacks,” Proc. of the IEEE Symp. on Research in Security and Privacy, pp. 72–84, May 1992.

[5] S.M. Bellovin, M. Merritt, “Augmented Encrypted Key Exchange: Password-based Protocols Secure Against Dictionary Attacks and Password File Compromise,” Proc. of the 1st ACM Conf. on Computer and Communications Security, pp. 244–250, Nov. 1993.

[6] Stainslaw Jarecki, Hugo Krawczyk, Jiayu Xu, “OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks,” Advances in Cryptology, Eurocrypt 2018, LNCS 10822, pp. 456–486, Springer, 2018.

[7] D. M’Raihi, M. Bellare, F. Hoornaert, D. Nccache, O. Ranen, “HOTP: An HMAC-based One-Time Password Alogrithm,” IETF RFC 42267, Dec. 2005.

[8] H. Krawczyk, P. Eronen, “HMAC-based Extract-and-Expand Key Derivation Function (HKDF),” IETF RFC 5869, May 2010.

[9] B. Kaliski, “PKCS #5: Password-Based Cryptography Specification Version 2.0,” IETF RFC 2898, May 2000.

[10] L. Lamport, “Password Authentication with Insecure Communication,” Communications of the ACM, Vol. 24. No. 11, pp. 770–772, Nov. 1981.

[11] N. Haller, C. Metz, P. Nesser, M. Straw, “A One-Time Password System,” IETF RFC 2289, Feb. 1998.

[12] Ari Juels, John Brainard, “Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks,” Proc. of NDSS ’99, pp. 151–165, 1999.

[13] Adam Back, Hashcash, http://www.cypherspace.org/hashcash/, 1997.

[14] Amos Fiat, “Batch RSA,” Advances in Cryptology, CRYPTO ’89, LNCS 435, pp. 175–185, Springer, 1990.

[15] R.C. Merkle, “A Digital Signature Based on a Conventional Encryption Function,” Advances in Cryptology, CRYPTO ’87, LNCS 293, pp. 369–378, Springer, 1988.

[16] L. Lamport, “Constructing Digital Signatures from a One way function,” Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.

[17] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, Markus Rückert, “On the Security of the Winternitz One-Time Signature Scheme,” Advances in Cryptology, ASIACRYPT ’91, LNCS 6737, pp. 363–378, Springer, 2011.

[18] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, A. Mohaisen, “XMSS: eXtended Merkle Signature Scheme,” IETF RFC 8391, May 2018.

[19] `http://sphincs.org`

# 퀴즈

1. 해시 기반 서명과 관련된 다음 설명 중 틀린 것은?

   ① 기본적으로 일회용 서명이다.

   ② 해시함수는 해시값의 길이만 늘리면 양자컴퓨팅이 현실화되어도 안전성에 문제가 없기 때문에 양자 내성 전자서명의 대안으로 검토되고 있다.

   ③ Winternitz OTS에서 w가 증가하면 서명과 서명 확인을 위해 필요한 해시 연산의 수와 공개키의 길이 (공개키를 나타내기 위해 필요한 해시값의 수)는 증가한다.

   ④ Winternitz OTS는 쉽게 존재 위조가 가능하므로 추가적인 checksum을 포함하여 서명해야 한다.

2. 패스워드 기반 암호키는 패스워드 추측 공격이 가능하므로 보통 로컬에서 장기간 키를 보호하기 위해 많이 사용한다. 하지만 EKE처럼 암호프로토콜을 적절하게 설계하면 패스워드 기반 키를 원격에 있는 사용자 간에도 사용할 수 있다. 암호프로토콜에서 패스워드 기반 암호키의 사용과 관련된 다음 설명 중 틀린 것은?

① 명백한 잉여 정보가 포함된 평문을 암호화하면 안 된다.

② 랜덤한 값만 암호화하더라도 그 값의 특성 때문에 여전히 패스워드 추측 공격에 취약할 수 있다.

③ 암호화된 랜덤값을 향후에 어떻게 사용하는지는 패스워드 추측 공격에 영향을 주지 않는다.

④ 패스워드 추측 공격을 하여 공격에 성공하였다는 것은 알 수 없지만 실패하였다는 것을 알 수 있으면 패스워드 추측 공격에 취약한 것이 된다.

3. Lamport의 해시체인과 해시체인을 이용한 일회용 패스워드와 관련된 다음 설명 중 틀린 것은?

① 해시함수의 일방향성을 이용하는 기법으로, 시작값을 반복적으로 해시하여 체인의 값을 생성한다.

② 해시체인은 생성한 순서대로 사용한다.

③ 해시체인의 루트를 서버에 등록하고 체인에 있는 값을 차례대로 사용하여 사용자를 인증할 수 있다. 이때 해시체인 루트 등록이 안전하게 이루어져야 하며, 등록된 루트값이나 현재 값을 확인하기 위한 값을 공격자가 교체할 수 없어야 한다.

④ 해시체인의 길이(반복적인 해시하여 생성한 해시값의 수)에 의해 인증 횟수가 제한된다.

4. 패스워드 해싱과 관련된 다음 설명 중 틀린 것은?

① 패스워드 자체 대신 패스워드의 해시값을 저장하면 패스워드를 보관한 파일이나 데이터베이스에 공격자가 접근하더라도 쉽게 패스워드가 노출되지 않는다.

② 사전공격을 통해 패스워드의 해시값으로부터 패스워드를 알아낼 수 있기 때문에 사전공격을 어렵개 하기 위해 해시값을 계산할 때 salt라고 하는 랜덤한 요소를 추가한다.

③ 사전공격을 통해 패스워드의 해시값으로부터 패스워드를 알아낼 수 있기 때문에 사전공격을 어렵개 하기 위해 해시값을 계산할 때 일반 해시함수 달리 계산 속도를 조절할 수 있는 해시함수를 사용한다.

④ 패스워드의 해시값을 계산할 때 사용하는 여러 요소 때문에 사전 공격이 힘들어졌으므로 쉬운 패스워드를 선택하여도 된다.

5. 동기화 방식의 OTP와 관련한 다음 설명 중 틀린 것은?

① 동기화 방식은 모두 사용자와 서버 간의 대칭키를 공유하고 있으며, 이 키를 이용하여 일회용 패스워드를 생성한다.

② 사건 동기화 방식은 사용자가 서버에 전달하지 않고 많이 생성하게 되면 사용자와 서버 간 카운터 값의 차이가 너무 커질 수 있어 동기화하는 것이 어려워질 수 있다.

③ 일회용 패스워드를 생성할 때 사용하는 키를 공격자가 획득하더라도 시간, 사건 기반 모두 미래에 사용할 값을 생성할 수 없다.

④ 시간 동기화 방식은 정해진 매 시간 간격마다 자동 생성된다.

## 연습문제

1. 패스워드 해싱에 대해 사전 공격을 취약할 수 있기 때문에 안전한 패스워드의 선택이 중요하다. 대칭 암호알고리즘에 대해서도 사전 공격이 가능한지 설명하시오. 대칭 암호알고리즘에 대한 사전 공격이란 가능한 모든 키로 특정 평문에 대한 암호문을 모든 구하여 표를 만들어 공격하는 것을 말한다.

2. Bcrypt처럼 패스워드 해시함수 알고리즘의 속도를 느리게 하였을 때 서비스 거부 공격이 쉬워질 수 있다. 그 이유를 설명하시오.

3. 패스워드 해싱에서 소금(salt)을 사용함으로써 얻어지는 두 가지 효과는 무엇인지 설명하시오.

4. 패스워드 해싱할 때 보통 소금을 추가하며, 이 소금값을 해시값과 함께 보관한다. $k$비트 소금값을 보관하지 않고 서버는 가능한 모든 $k$비트를 이용하여 해시값을 확인하도록 할 수 있다. 물론 계산 비용 때문에 긴 길이를 사용할 수 없지만 성능에 문제가 안 되는 길이(예: 16bit)를 사용한다면 사전 공격에 도움이 되는지 논하고, 이 방법을 사용하면 발생할 수 있는 다른 문제점은 없는지 논하시오.

5. 우리는 인터넷 뱅킹 등을 사용하기 위해 공개키 인증서를 발급받아 사용한다. 이를 위해 모바일 폰이나 USB에 공개키 인증서를 보관하여 사용한다. 이때 공개키는 인증서 형태로 유지되지만 개인키는 패스워드 기반 암호화 방식을 이용하여 암호화되어 있다. 이때에도 패스워드로부터 대칭키를 생성하기 위해 salt를 사용한다. 이 salt는 어디에 보관되는지 설명하시오.

6. 패스워드 기반 프로토콜은 패스워드를 이용하여 만든 암호키를 통신 메시지에 직접 사용하는 프로토콜이다. 이 경우 이와 같은 암호키는 랜덤한 값(명백한 잉여정보가 아닌)만 암호화하여야 한다. 하지만 이것만으로는 패스워드 추측 공격에 강건하지 못하다. 랜덤한 값을 암호화하더라도 추가로 문제가 될 수 있는 상황을 몇 가지 설명하시오.

7. 10장에서 살펴본 무선랜 보안 프로토콜 중 WPA3은 패스워드 기반 프로토콜을 사용한다. WPA3는 기본적으로 패스워드 기반 대칭키를 무선 장치와 무선 AP 간에 사용하므로 패스워드 기반 프로토콜을 사용하는 것이 필요하다. 그런데 패스워드 기반 프로토콜을 사용한다고 모든 보안 문제가 해결되는 것은 아니다. 무선랜 환경에서 패스워드 기반 프로토콜을 사용하면 어떤 공격을 방어할 수 있고, 어떤 공격은 방어할 수 없는지 설명하시오.

8. OTP(One-Time Password)는 매번 다른 패스워드를 사용하도록 하여 패스워드의 안전성을 높이는 기술이다. 현재 인터넷 뱅킹에서 사용되고 있으며, 하드웨어 장치를 사용한다. 하드웨어 장치를 사용하는 이유를 설명하시오.

9. OTP에서 시간 동기화 방식을 간단히 설명하고, 그것의 문제점을 설명하시오.

10. $M_1$부터 $M_n$까지 $n$개 메시지가 있을 때, 이들을 해시 트리를 표현하여 루트에 서명하는 것, $H(M_1||M_2||\dots||M_n)$에 서명하는 것, $H(H(M_1)||H(M_2)||\dots||H(M_n))$에 서명하는 것과 비교하시오.

11. Wintermitz OTS의 공개키를 여러 개 생성하여 이들의 해시값이 단말이 되는 해시 트리를 구성하여 일회용 전자서명으로 다중 전자서명으로 확장할 수 있다. 예를 들어 SHA-256을 이용하여 Wintermitz OTS를 구현하였고, 이때 $w = 16$을 사용하였다고 하자. 또한 8번 서명할 수 있도록 Wintermitz OTS 공개키 8개를 생성하여 해시 트리를 구성하였다고 하자. 이 트리 모습을 제시하고 3번째 키를 이용하여 어떤 메시지에 서명하였을 때, 서명 값으로 제시해야 하는 것을 나열하시오.