Key Management

2025.08

자동차융합대학

GENERAL MOTORS
GM TECHNICAL CENTER KOREA

KMU 국민대학교
KOOKMIN UNIVERSITY

CONTENTS

01

암호학적 난수

암호학적 난수

■ 난수 생성기 (True Random Number Generator, TRNG)

- 비예측성 & 비결정성
- 전자 저항에서 생성되는 열 잡음의 표본을 추출하거나, 방사선 관측기로부터 나오는 출력 값을 반복해서 사용 → 환경적 제약

■ 의사 난수 생성기 (Pseudo Random Number Generator, PRNG)

- 초기 값(Seed)을 입력 받아 계산되는 의사 난수열을 출력하며, 같은 입력 값에 대하여 같은 출력 값을 생성하는 결정적(Deterministic) 알고리즘 → 통계적 검증

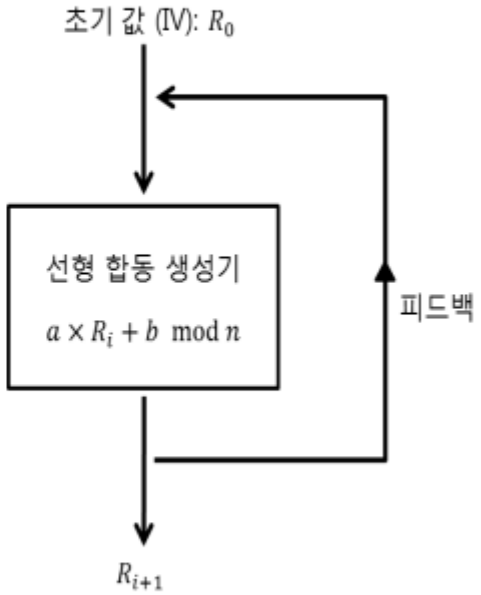
	생성시간	결정성	주기성
TRNG	비효율적	비결정적	비주기적
PRNG	효율적	결정적	주기적

암호학적 난수 (cont'd)

■ 선형 합동 생성기 (Linear Congruential Generator)

- $R_{i+1} \equiv (a \times R_i + b) \bmod n,\ 0 \leq R_0 < n$: 초기 값
- a, b를 비밀키로 공유, 법 n은 공개
- $a = 3, b = 2, n = 17$

i	R_i	$a \times R_i + b$	R_{i+1}	i	R_i	$a \times R_i + b$	R_{i+1}
0	8	$3 \times 8 + 2$	9	10	3	$3 \times 3 + 2$	11
1	9	$3 \times 9 + 2$	12	11	11	$3 \times 11 + 2$	1
2	12	$3 \times 12 + 2$	4	12	1	$3 \times 1 + 2$	5
3	4	$3 \times 4 + 2$	14	13	5	$3 \times 5 + 2$	0
4	14	$3 \times 14 + 2$	10	14	0	$3 \times 0 + 2$	2
5	10	$3 \times 10 + 2$	15	15	2	$3 \times 2 + 2$	8
6	15	$3 \times 15 + 2$	13	16	8	$3 \times 8 + 2$	9
7	13	$3 \times 13 + 2$	7	17	9	$3 \times 9 + 2$	12
8	7	$3 \times 7 + 2$	6	18	12	$3 \times 12 + 2$	4
9	6	$3 \times 6 + 2$	3	⋯			



암호학적 난수 (cont'd)

■ 선형 합동 생성기 (Linear Congruential Generator)

- 최대 주기를 갖기 위해서는 다음과 같은 조건을 고려
  - 모듈로 n의 크기에 따라 최대 주기가 결정되기 때문에, 효율성과 안전성을 고려하여 n을 선택
    - ✓ 보통 컴퓨터가 한번에 데이터를 처리할 수 있는 워드의 크기 만큼을 n으로 선택함
  - n의 크기와 같은 주기를 갖기 위해서는 a를 n과 서로소인 수로 선택
  - b의 값은 주기에 영향을 미치진 않지만, 계산의 효율성을 위해 일반적으로 0으로 사용

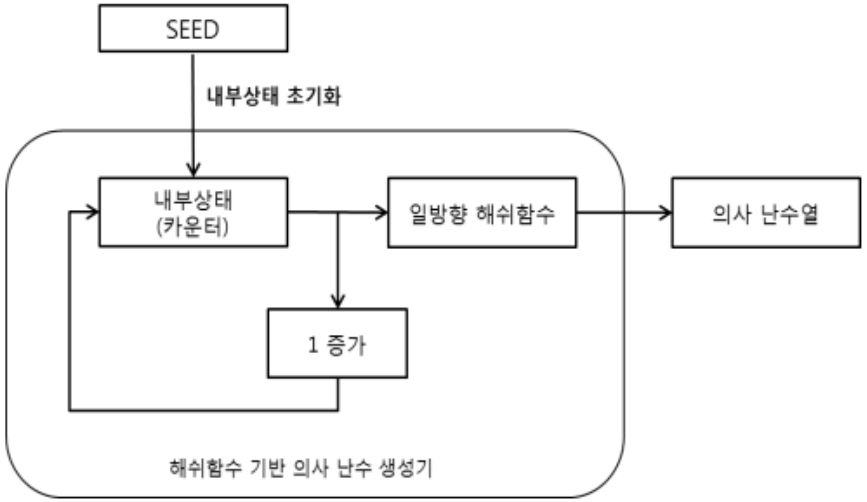
■ 암호학적으로 안전한 의사 난수 생성기 (Cryptographically Secure PRNG, CSPRNG)

- 예측 불가능성이 확보되어야 함
- 난수 생성방법
  - *일방향 해쉬 함수*
  - *블록 암호*
  - 수학적 난제에 기반한 생성 방법

암호학적 난수 (cont'd)

■ 일방향 해쉬 함수를 사용한 의사 난수 생성기

- 일방향 함수 $f(x) = y$
  - 주어진 x에 대하여 함수 $f(\cdot)$를 이용하여 y를 계산하는 것은 쉽지만 반대로 y가 주어졌을 때 함수 $f(\cdot)$를 이용하여 $\mathrm{x} = \mathrm{f}^{-1}(y)$를 계산하는 것은 어려운 함수
  - $p, q : prime$
  - $g(p, q) = p \times q$

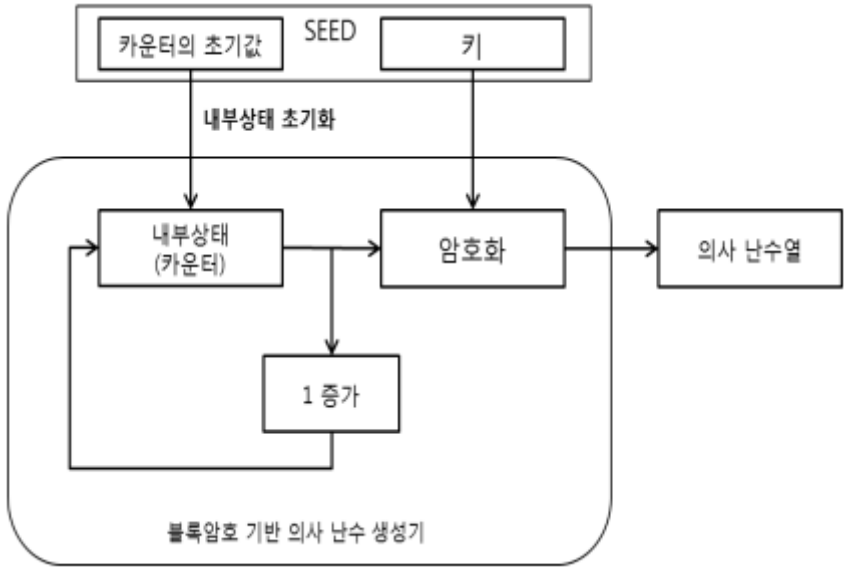


- 해쉬함수 $f(x) = y$
  - 상이한 입력 값에 고정된 길이의 출력 값
  - 압축함수
  - 해쉬 함수의 일방향성에 의하여 해당 난수에 해당하는 카운터 값을 예상할 수 없음
  - 카운터 값을 알 수 없다면, 앞으로 생성될 난수열이나 이전에 생성된 난수열을 예측할 수 없게 되므로 CSPRNG 조건을 만족함

암호학적 난수 (cont'd)

■ 암호를 사용한 의사 난수 생성기

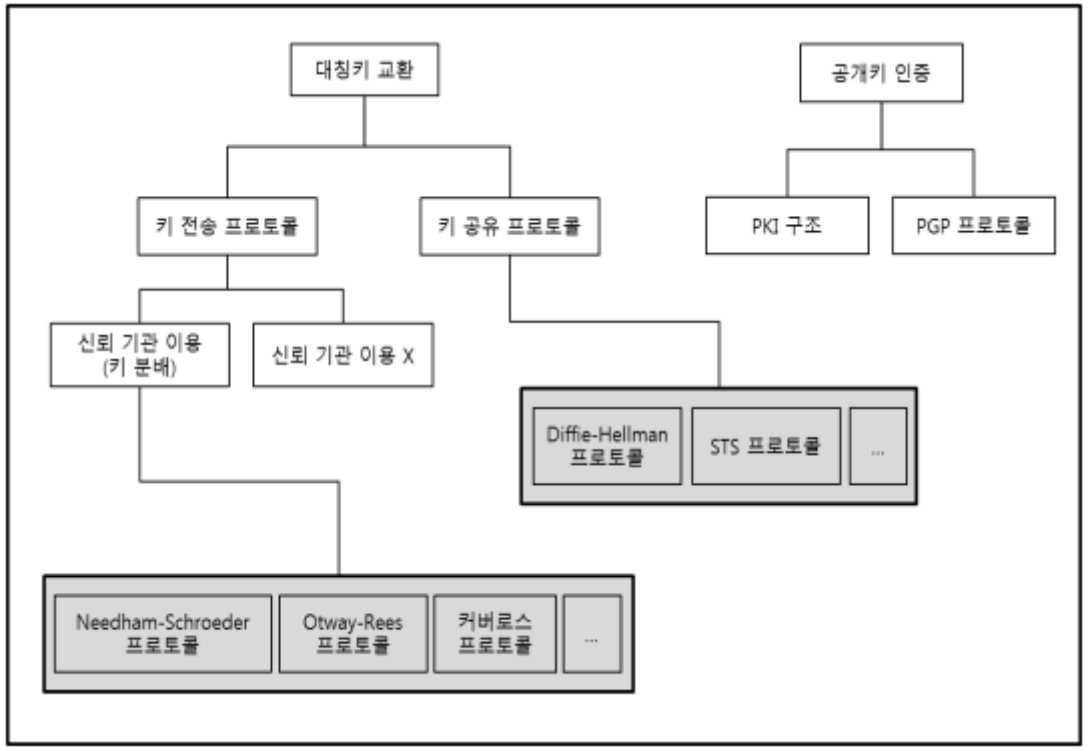
- 암호 알고리즘을 이용하기 때문에 의사 난수열을 보고 카운터를 예상할 수 없음



02

Key Distribution

키 관리 방법


대칭키 교환
공개키 인증
키 전송 프로토콜
키 공유 프로토콜
PKI 구조
PGP 프로토콜
신뢰 기관 이용
(키 분배)
신뢰 기관 이용 X
Diffie-Hellman
프로토콜
STS 프로토콜
...
Needham-Schroeder
프로토콜
Otway-Rees
프로토콜
커버로스
프로토콜
...

키 분배 (Key Distribution)

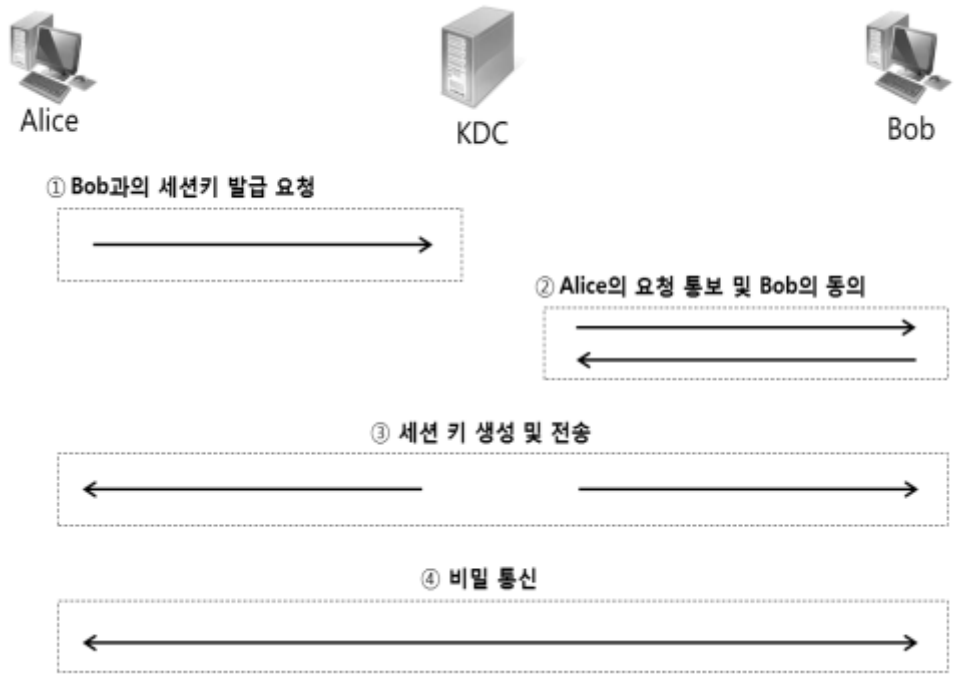
■ 대칭키를 이용한 키 분배

- n명 : $\frac{n(n-1)}{2}$ 키 필요, 사용자는 $(n-1)$ 관리
- 제 3자인 키 분배 센터(KDC)를 이용

Key Distribution (cont'd)

■ 대칭키를 이용한 키 분배

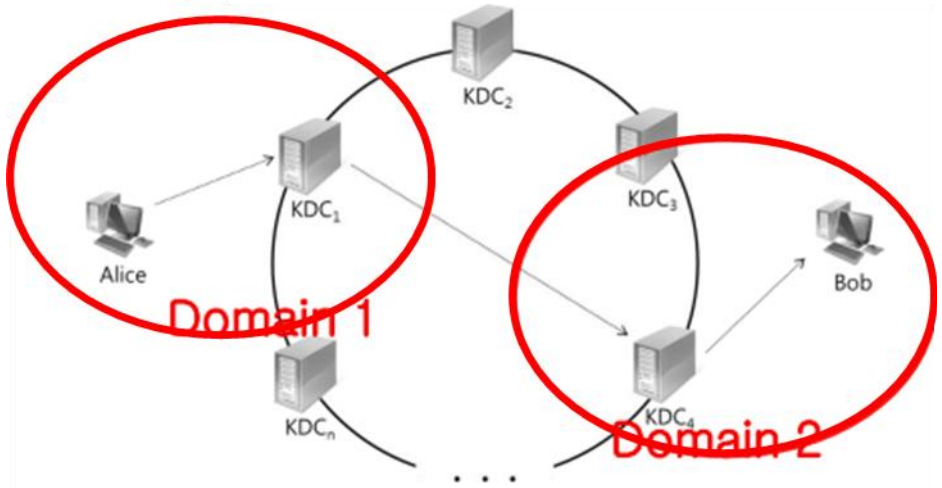
- 키 분배 센터를 이용한 키(세션키) 분배 방법



Key Distribution (cont'd)

■ 대칭키를 이용한 키 분배

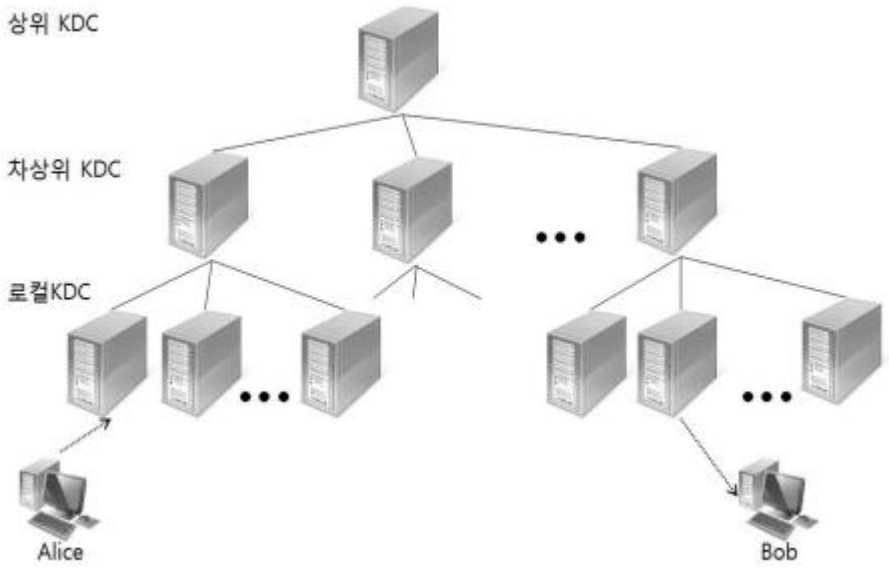
- 분산된 키 분배 센터를 이용한 키 분배 방법
  - 평등 다중(Flat Multiple) 구조의 키 분배 센터
    - ✓ Alice → KDC_1 : 세션키 생성 요청
    - ✓ KDC_1 → KDC_4 : Alice의 요청 전달
    - ✓ KDC_4 → Bob : Alice의 요청 알림
    - ✓ Bob → KDC_4 : 동의
    - ✓ KDC_4 → KDC_1 : Bob의 동의 알림
    - ✓ KDC_1 → KDC_4 : 세션키(K_{AB}) 전송
    - ✓ KDC_1 → Alice : $E_{SK_A}(K_{AB})$
    - ✓ KDC_4 → Bob : $E_{SK_B}(K_{AB})$



Key Distribution (cont'd)

■ 대칭키를 이용한 키 분배

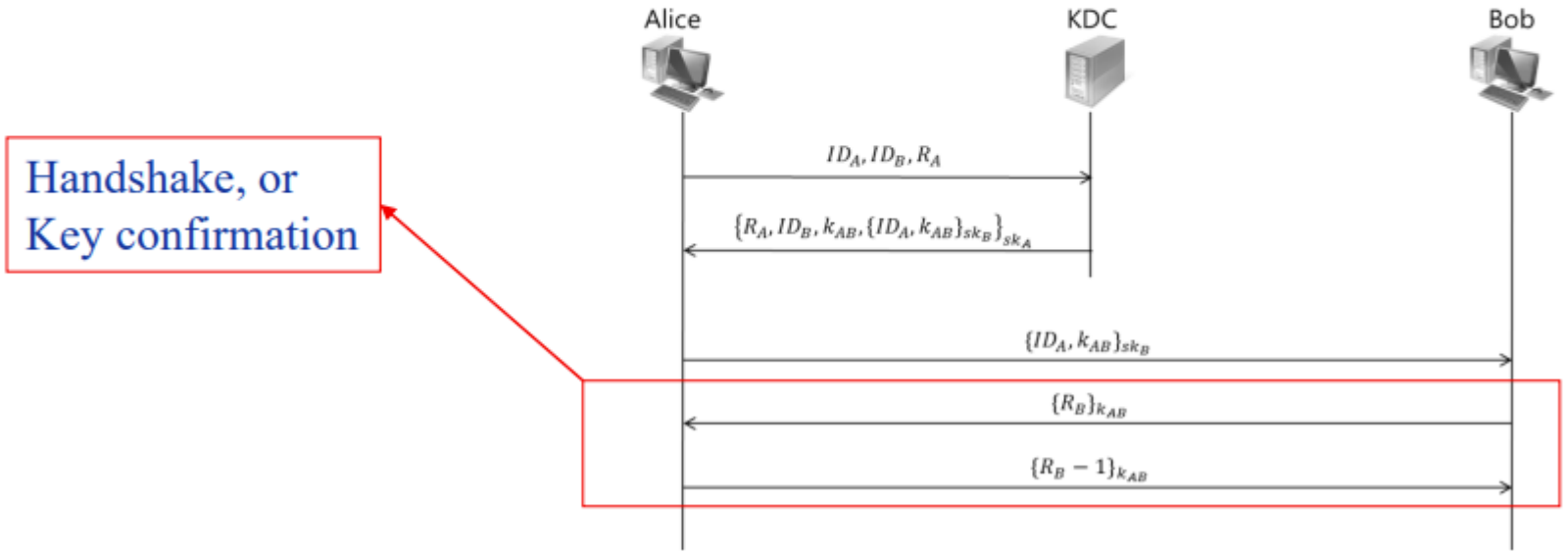
- 분산된 키 분배 센터를 이용한 키 분배 방법
  - 계층 다중(Hierarchical Multiple) 구조의 키 분배



Key Distribution (cont'd)

■ **Needham-Schroeder 프로토콜**

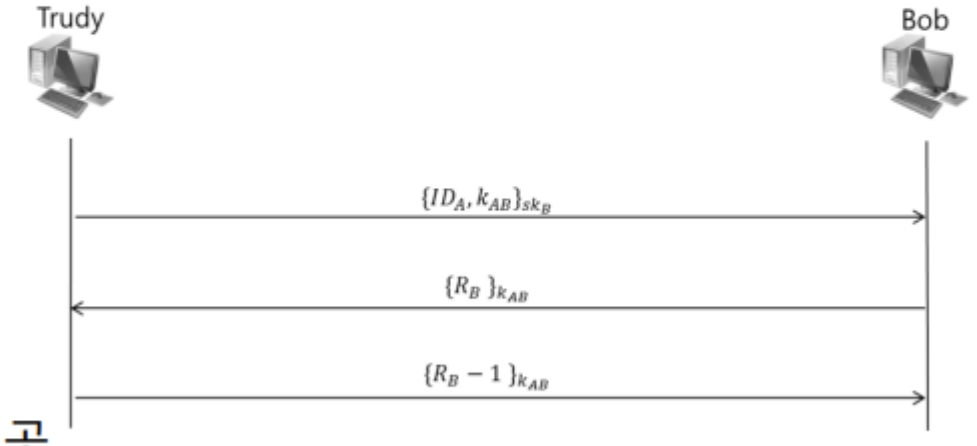
1. Alice → KDC : ID_A, ID_B, R_A
2. KDC → Alice : $E_{sk_A}(R_A, ID_B, k_{AB}, E_{sk_B}(ID_A, k_{AB}))$
3. Alice → Bob : $E_{sk_B}(ID_A, k_{AB})$
4. Bob → Alice : $E_{k_{AB}}(R_B)$
5. Alice → Bob : $E_{k_{AB}}(R_B - 1)$



Key Distribution (cont'd)

■ **Needham-Schroeder 프로토콜 재전송 공격**

- Trudy with old session key k_{AB}
  3. Trudy → Bob : $E_{sk_B}(ID_A, k_{AB})$
  4. Bob → Alice(Trudy) : $E_{k_{AB}}(R_B)$
  5. Trudy → Bob : $E_{k_{AB}}(R_B - 1)$



- 재전송 공격 방지
  - 세션키에 새로움 제공

→ $E_{sk_B}(ID_A, k_{AB}, T)$ in Step 2

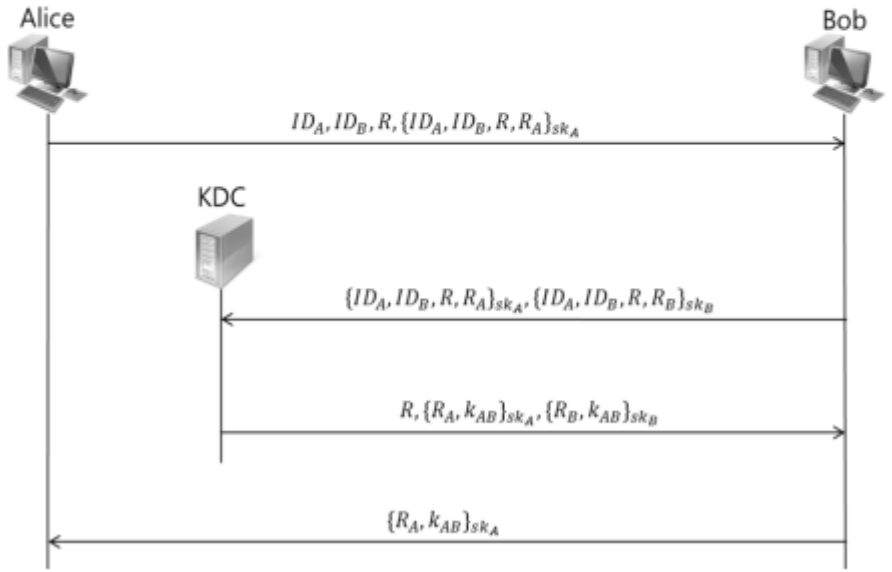
Key Distribution (cont'd)

■ **Otway-Rees 프로토콜**

1. Alice → Bob : $ID_A,\ ID_B,\ R,\ E_{sk_A}(ID_A, ID_B, R,\ R_A)$
2. Bob → KDC : $E_{sk_A}(ID_A, ID_B, R, R_A),\ E_{sk_B}(ID_A, ID_B, R, R_B)$
3. KDC → Bob :$(R, E_{sk_A}(R_A, k_{AB}), E_{sk_B}(R_B, k_{AB}))$
4. Bob → Alice : $E_{sk_A}(R_A, k_{AB})$

- R : Index number
- R_A : Alice 확인
- R_B : Bob 확인

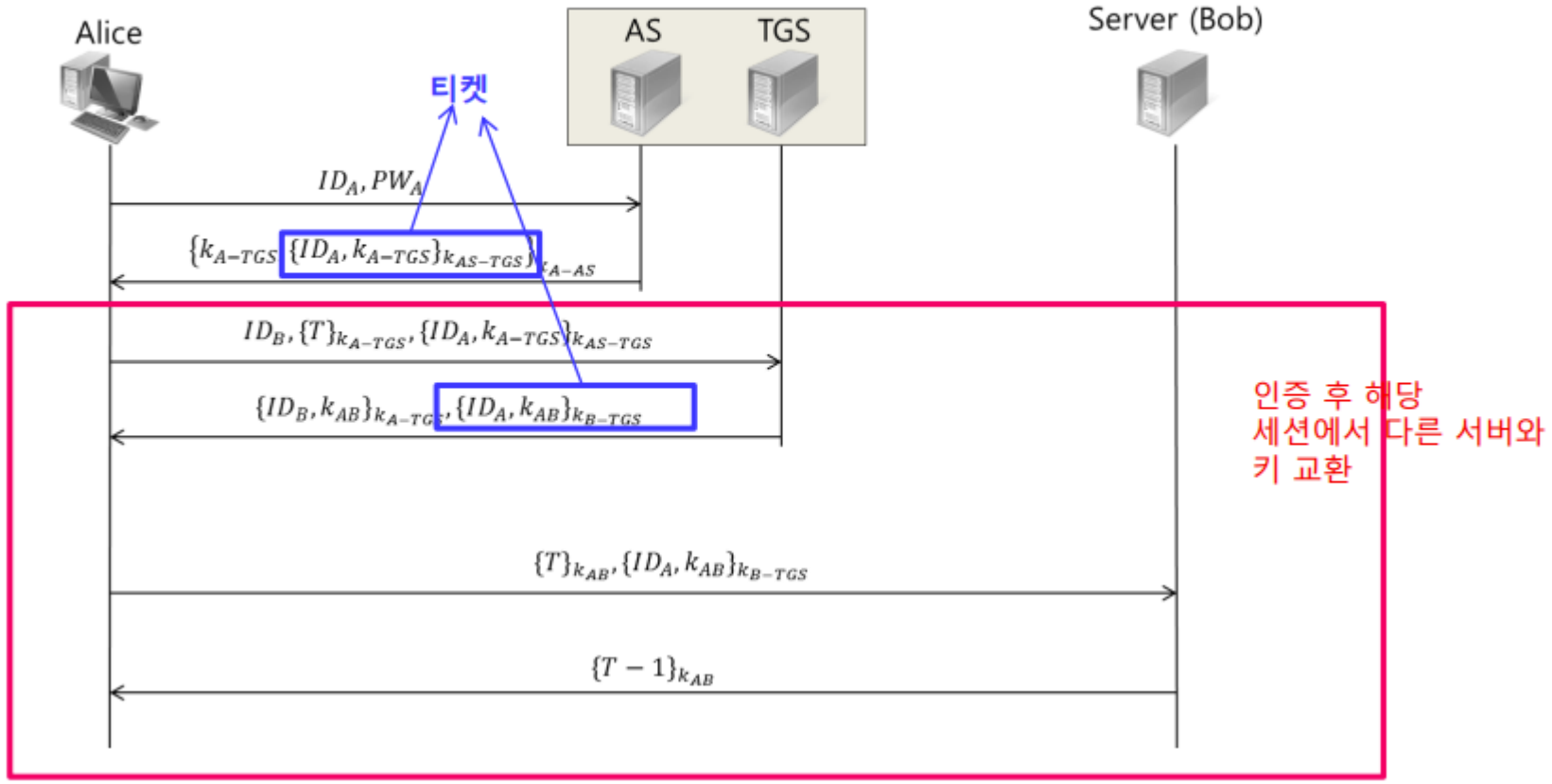
✘ 재전송 공격?



Key Distribution (cont'd)

■ 커버로스 (Kerberos)

- MIT에서 네트워크 내부 사용자 인증

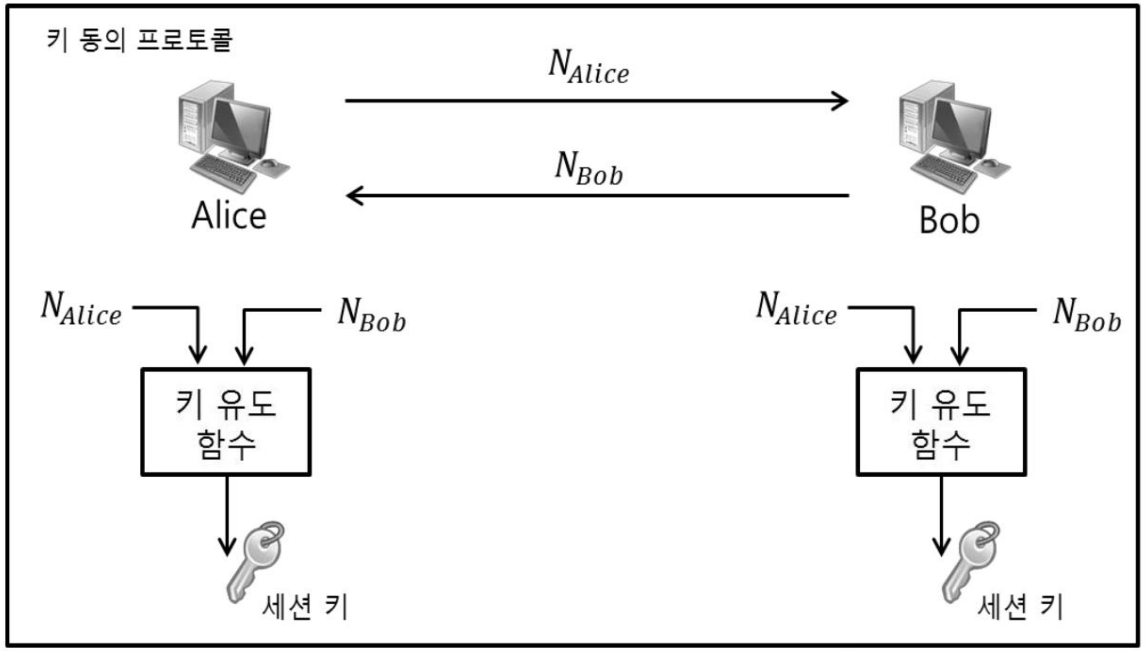


03

Key Exchange or Establishment

❖ 키 전송 (Key Transport) 프로토콜

❖ 키 동의 (Key Agreement) 프로토콜



Key Exchange (cont'd)

■ 키 교환 프로토콜의 안전성

- 전방향 안전성 (Forward Secrecy)
  - 사용자의 비밀키를 알고 있는 공격자라도 정직한 구성원 간에 성공적으로 확립된 이전의 세션키에 대한 어떠한 정보도 얻을 수 없어야 함

1. Alice → Bob : $E_{pk_B}(k_n)$
2. Eve : $\{E_{pk_B}(k_1), E_{pk_B}(k_2), \ldots, E_{pk_B}(k_n)\}$ 저장 & pk_B 노출

→ 이전 세션의 정보가 누출

- 기지-키 안전성 (Known-Key Secrecy)
  - 여러 세션에서 얻은 세션키들을 이용해도 노출되지 않은 세션키들의 기밀성에는 영향을 주지 않아야 함

1. 새로운 세션키 $k = h(k', ID_A, ID_B)$
2. Eve : 세션키 $k' \rightarrow k$ 계산

→ 이전 세션키를 이용해 이후의 세션키 도출 가능

Key Exchange (cont'd)

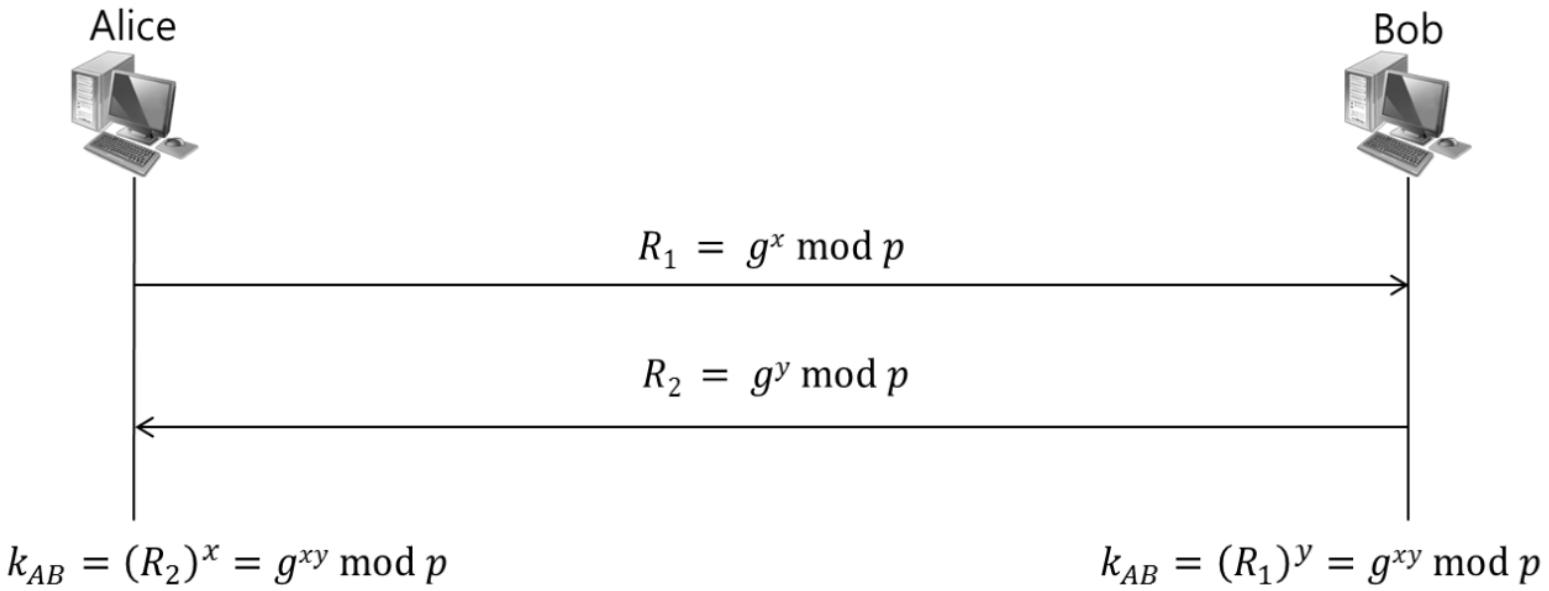
■ 키 교환 프로토콜의 안전성

- 세션 상태 노출에 대한 안전성(Security against Session State Reveal)
  - 공격자가 세션키를 만드는 데 사용되는 난수 값을 가지고서도 세션키를 알 수 없어야 함
  - 롱텀키(long-term key)인 비밀키 보다는 일회용 비밀 값인 난수들이 더욱 쉽게 노출될 수 있다는 관점

- 비밀키 사용 위장에 대한 안전성(Security against Key Compromise Impersonation)
  - Eve가 Alice의 비밀키로 Bob으로 위장함을 방지

- 파트너 혼돈 공격에 대한 안전성(Security against Unknown Key Share)
  - Alice와 Bob이 동일한 세션키를 계산했다면 Alice는 현재 Bob과 키 교환을 하고 있다고 인식해야 하며, Bob 또한 Alice와 키 교환을 하고 있다고 인식해야 함

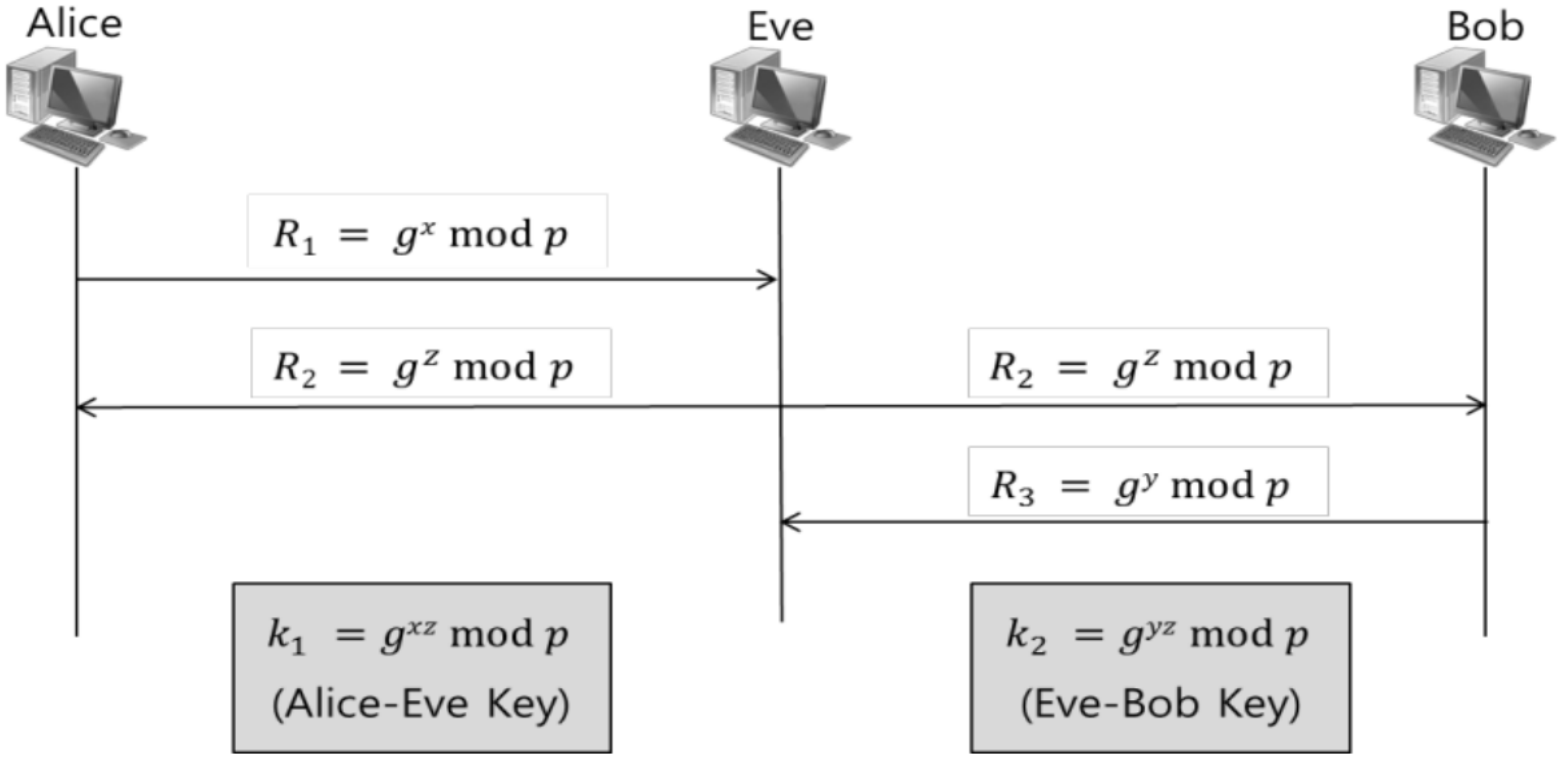
■ Diffie-Hellman 동의 프로토콜



Key Exchange (cont'd)

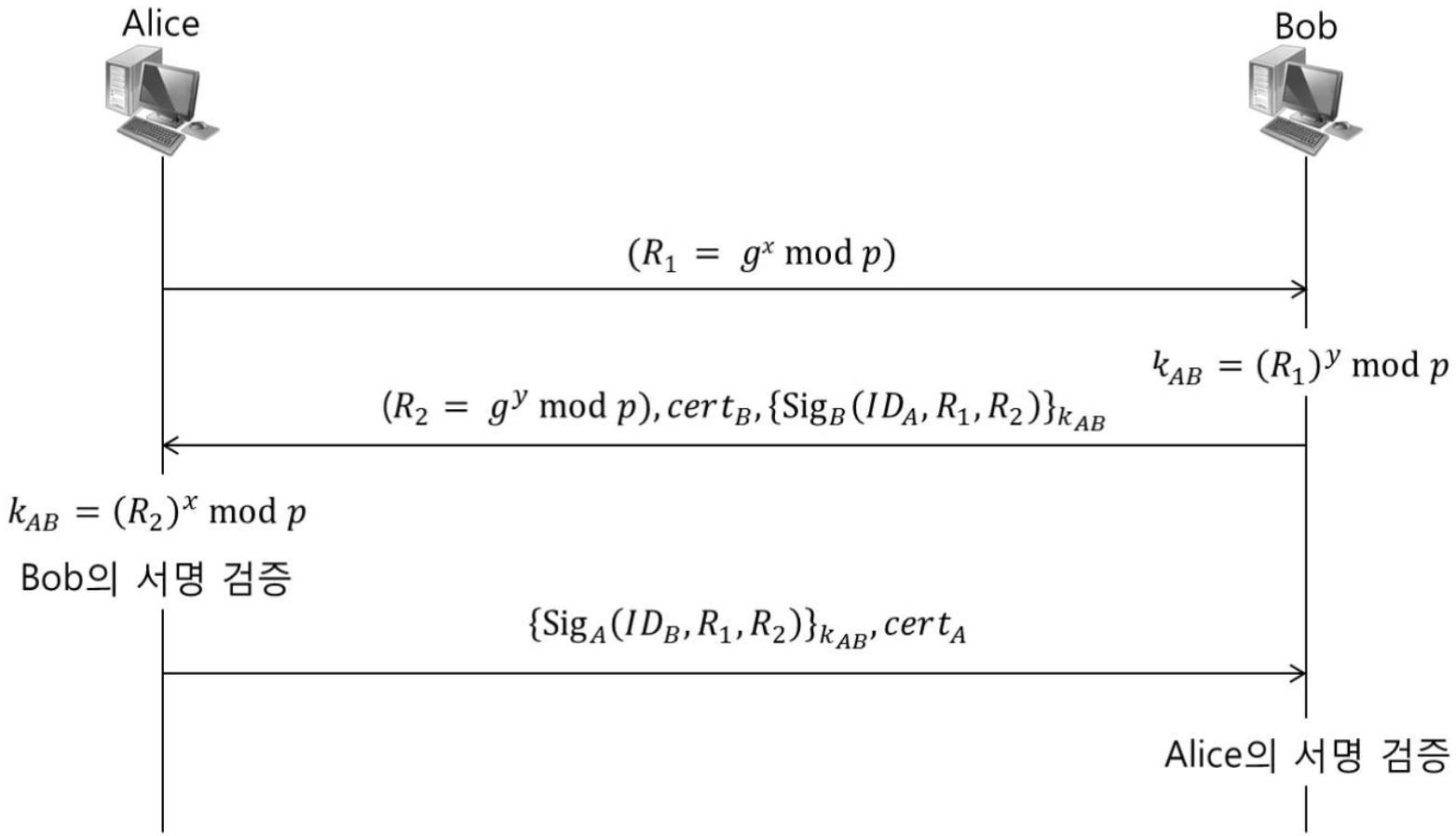
■ Diffie-Hellman 동의 프로토콜

- 중간자 공격 (MIMT, Man-in-the-Middle-Attack)



$R1$ 이 Alice의 인증서, $R2$가 Bob의 인증서인 경우?

■ STS (Station-To-Station) 프로토콜



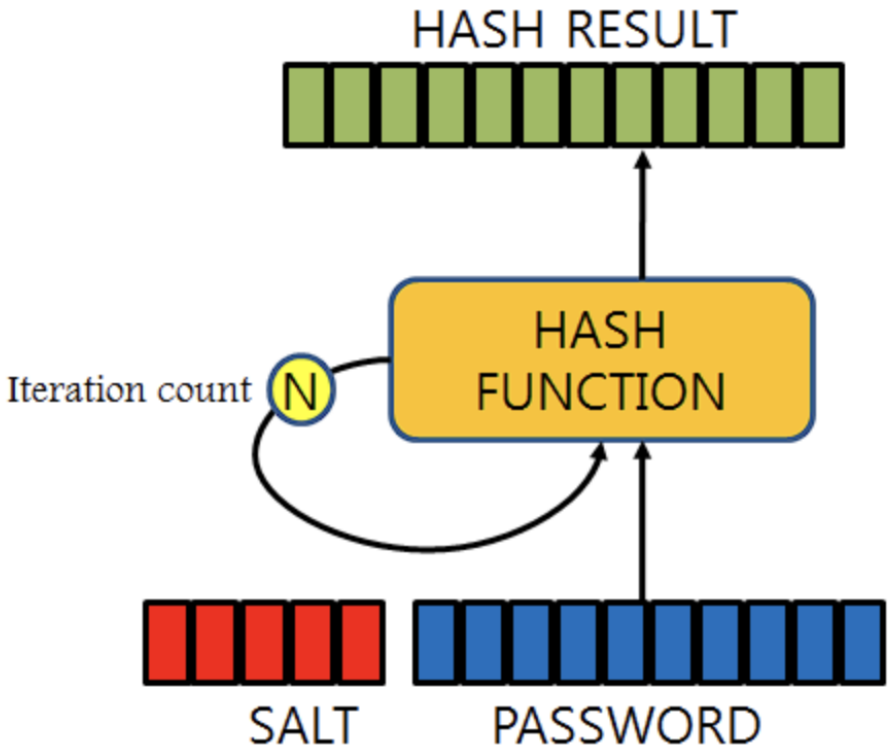
04

Key Derivation Function

Key Derivation Function

■ Adaptive Key Derivation Function

- Adaptive key derivation function은 digest를 생성할 때 salting과 key stretching을 반복하여 공격자가 쉽게 digest를 유추할 수 없도록 하고 보안의 강도를 설정할 수 있게 한다.
- Salt
  - 패스워드에 추가하는 임의의 문자열로, 최소 128bits 정도는 되어야 안전하다.
  - Salting된 digest로는 해커가 패스워드 일치 여부를 알기 매우 어려우며, 사용자마다 다른 salt를 사용한다면 패스워드가 같더라도 digest가 다르게 생성된다.
- Key Stretching
  - 해시를 여러 번 반복하여 계산 시간을 충분히 늘려 Brute force attack에 대비할 수 있다.

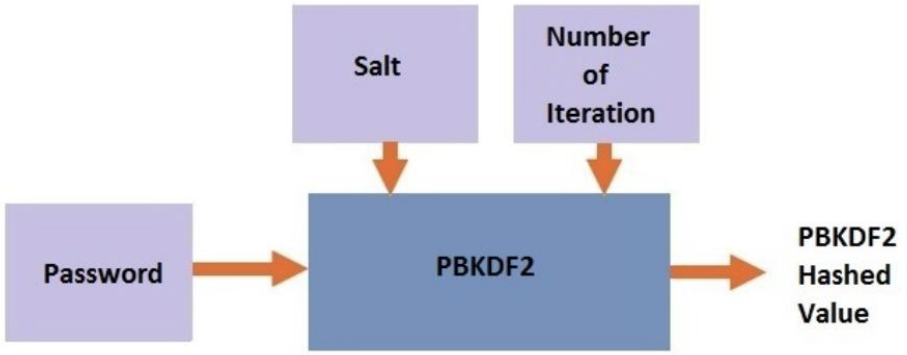


■ Password Based KDF2 (PBKDF2)

- PBKDF2는 NIST(미국표준기술연구원)에 의해서 승인된 알고리즘으로, 미국 정부 시스템에서도 사용자 패스워드와 암호화된 digest를 생성할 때 사용한다.
- ISO-27001의 보안 규정을 준수하고, 3rd party의 라이브러리에 의존하지 않으면서 사용자 패스워드의 digest를 생성하려면 PBKDF2-HMAC-SHA-256/SHA-512을 사용하면 된다.

DIGEST = PBKDF2(PRF, Password, Salt, c, DLen)

- PRF: 난수(예: HMAC)
- Password: 패스워드
- Salt: 암호학 솔트 (32비트 이상 추천)
- c: 원하는 iteration 반복 수 (1000회 이상 추천)
- DLen: 원하는 다이제스트 길이



■ PBKDF2 (cont'd)

The PBKDF2 key derivation function has five input parameters:[9]

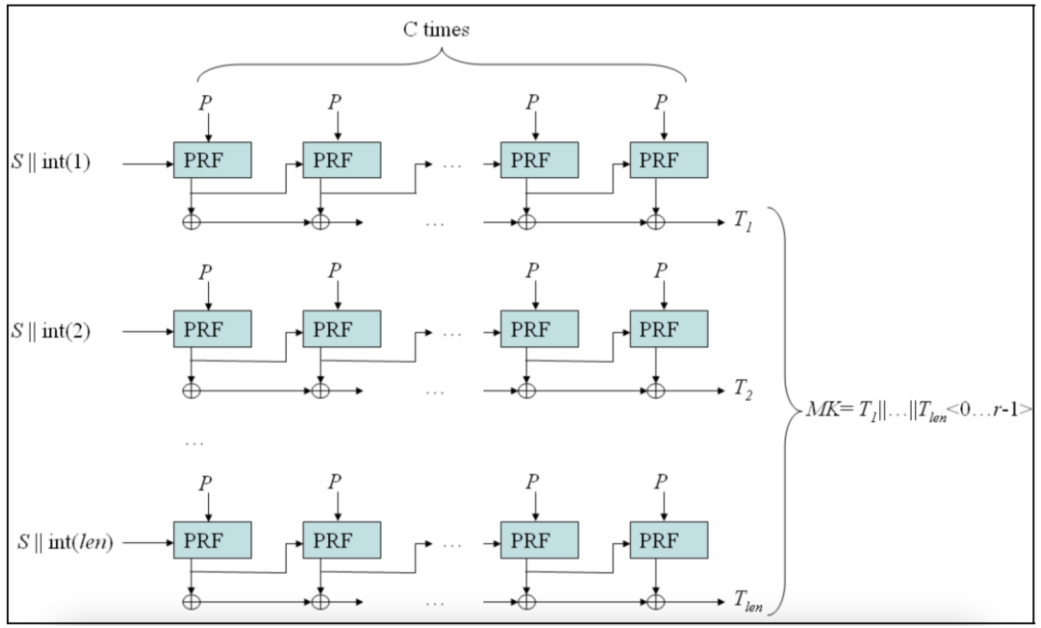
$$\mathrm{DK} = \mathrm{PBKDF2}(\mathrm{PRF}, \mathit{Password}, \mathit{Salt}, c, \mathit{dkLen})$$

where:

- PRF is a pseudorandom function of two parameters with output length *hLen* (e.g., a keyed HMAC)
- *Password* is the master password from which a derived key is generated
- *Salt* is a sequence of bits, known as a cryptographic salt
- *c* is the number of iterations desired
- *dkLen* is the desired bit-length of the derived key
- *DK* is the generated derived key

Each *hLen*-bit block T_i of derived key *DK*, is computed as follows (with + marking string concatenation):

$$DK = \mathrm{T}_1 + \mathrm{T}_2 + \cdots + \mathrm{T}_{dklen/hlen}$$
$$\mathrm{T}_i = \mathrm{F}(\mathit{Password}, \mathit{Salt}, c, i)$$



$$\mathrm{F}(\mathit{Password}, \mathit{Salt}, c, i) = \mathrm{U}_1 \wedge \mathrm{U}_2 \wedge \cdots \wedge \mathrm{U}_c$$

where:

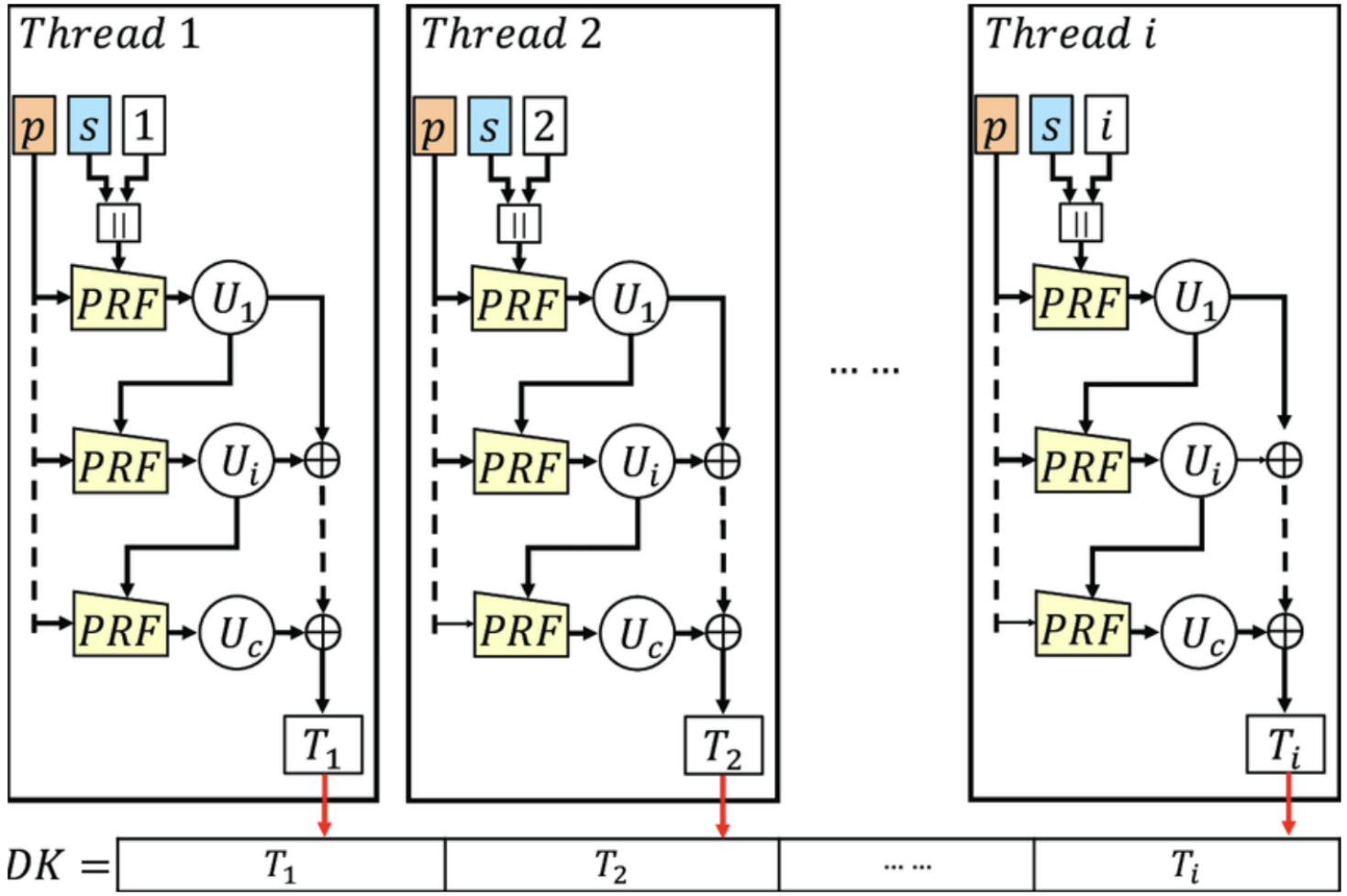
$$\mathrm{U}_1 = \mathrm{PRF}(\mathit{Password}, \mathit{Salt} + \mathrm{INT\_32\_BE}(i))$$
$$\mathrm{U}_2 = \mathrm{PRF}(\mathit{Password}, \mathrm{U}_1)$$
$$\vdots$$
$$\mathrm{U}_c = \mathrm{PRF}(\mathit{Password}, \mathrm{U}_{c-1})$$

For example, WPA2 uses:

$$\mathrm{DK} = \mathrm{PBKDF2}(\mathrm{HMAC{-}SHA1}, \mathit{passphrase}, \mathit{ssid}, 4096, 256)$$

[Ref] https://en.wikipedia.org/wiki/PBKDF2

■ PBKDF2 (cont'd)



■ Python Code of PBKDF2

```
def pbkdf2(digestmod, password: 'bytes', salt, count, dk_length) -> 'bytes':
    def pbkdf2_function(pw, salt, count, i):
        # in the first iteration, the hmac message is the salt
        # concatinated with the block number in the form of \x00\x00\x00\x01
        r = u = hmac.new(pw, salt + struct.pack(">i", i), digestmod).digest()
        for i in range(2, count + 1):
            # in subsequent iterations, the hmac message is the
            # previous hmac digest. The key is always the users password
            # see the hmac specification for notes on padding and stretching
            u = hmac.new(pw, u, digestmod).digest()
            # this is the exclusive or of the two byte-strings
            r = bytes(i ^ j for i, j in zip(r, u))
        return r
    dk, h_length = b'', digestmod().digest_size
    # we generate as many blocks as are required to
    # concatinate to the desired key size:
    blocks = (dk_length // h_length) + (1 if dk_length % h_length else 0)
    for i in range(1, blocks + 1):
        dk += pbkdf2_function(password, salt, count, i)
    # The length of the key wil be dk_length to the nearest
    # hash block size, i.e. larger than or equal to it. We
    # slice it to the desired length befor returning it.
    return dk[:dk_length]
```

[Ref] https://github.com/sfstpala/python3-pbkdf2/blob/master/pbkdf2.py

Thank You

MOBILITY CYBERSECURITY
MoSE