14. Analyzing a Cryptographic Protocol

온라인 서비스에서는 사용자와 서버가 서로를 **인증**해야 하며, 오프라인과 달리 메시지 위조, 도청, 피싱 사이트 등 다양한 위험이 존재한다. 이를 해결하기 위해 **인증 프로토콜**이 사용된다.

이 장에서는 상호 인증을 위해 제안된 유명한 프로토콜인 **Needham–Schroeder 공개키 인증 프로토콜(NSPK)**을 분석한다. 분석 방식은 복잡한 공격 기법을 찾는 것이 아니라, 시스템에 **침입자 모델**을 추가해 침입자가 취할 **모든 가능한 행동을 모델링하는 '브루트 포스' 방식**을 사용한다. 만약 이 모든 경우에도 프로토콜이 안전하다면, 실제로도 안전하다고 볼 수 있다는 취지다.

14.1 Public-Key Cryptography

1. **각 사용자(에이전트)는 두 개의 키를 가진다.**
   - **공개키(PK)**: 모두가 알고 있는 키
   - **개인키(PrvK)**: 오직 자신만 알고 있는 키
   - 공개키로 암호화한 데이터는 대응되는 개인키로만 복호화 가능, 그 반대도 동일함.
2. **공개키 암호의 장점**
   - 서로 **공유한 비밀키 없이도** 두 사용자가 안전하게 통신 가능.
   - 예: Alice가 Bob에게 메시지를 보내려면 Bob의 공개키로 암호화 → Bob만 복호화 가능.
3. **하지만 인증 문제는 해결되지 않음**
   - Bob은 메시지를 '누가' 보냈는지 확인할 수 없음 (모두가 Bob의 공개키로 암호화된 메시지를 만들 수 있기 때문)
4. **공개키 암호의 목표**
   - 개인키를 역으로 추측하는 것이 계산적으로 불가능해야 함
   - 복호화 키 없이 암호문을 해독할 수 없어야 함
5. **RSA 알고리즘 개요**
   - 매우 큰 두 소수 p, q 선택 → 곱 n = p · q는 공개키의 일부
   - RSA의 안전성 = 큰 수 n의 소인수분해가 불가능하다는 점에 기반
   - 암/복호화 방식:

$$\text{암호화: } m^e \mod n$$
$$\text{복호화: } m^d \mod n$$

   - e: 공개키 지수, d: 개인키 지수(소인수분해로만 찾을 수 있음)

14.1.1 Digital Signatures

- 디지털 서명은 현실의 '문서 서명'과 같은 역할을 하며, **문서의 작성자·열람자 확인** 및 **위조 방지**를 가능하게 한다.
- 공개키 암호를 사용하면 디지털 계약문에 서명할 수 있다.

- 예: Peter가 계약문 m에 서명하고 은행에 보내고 싶다면,
  - 자신의 **개인키(PrvK)**로 m을 암호화하여 {m}_PrvK 형태로 전송한다.
  - 은행은 Peter의 **공개키(PK)**로 복호화해 실제 서명 여부를 확인한다.
- 이 방식은:
  1. 계약에 동의했다는 사실을 은행이 증명할 수 있게 하고
  2. Peter나 은행 중 누구도 계약을 나중에 위조하지 못하게 만든다.
- 전체 메시지를 암호화하는 비용이 크기 때문에 실제로는 보통 **해시 값 h(m)**을 개인키로 암호화한 {h(m)}_PrvK을 함께 보내는 방식 사용.

14.1.2 Symmetric-Key Cryptography

- 공개키 암호는 **연산 비용이 매우 커서 느림** → 대용량 데이터 암호화에 비효율적.
- 대칭키 암호는 두 사용자가 **같은 비밀키(secret key)**를 공유하고 이를 사용해 매우 빠르게 암·복호화함. (AES, DES는 RSA보다 **수백~수천 배** 빠름)
- 문제: 대칭키 암호를 사용하려면 **먼저 서로를 인증하고, 비밀키를 공유**해야 함.
- 해결: 공개키 암호를 사용해 안전한 채널을 먼저 만들고, 그 채널로 공유 비밀키를 교환한 뒤, 이후 통신은 대칭키로 진행.
- 이 구조는 TLS(HTTPS) 등 실제 인터넷 보안 프로토콜에서 사용됨.

14.2 The Needham-Schroeder Public-Key (NSPK) Protocol

NSPK 프로토콜의 목적

두 주체 A와 B가 **서로의 신원을 확인하며**, 안전하게 세션을 시작할 수 있도록 하는 공개키 기반 인증 프로토콜이다. 이 과정에서 **nonce(새로 생성된 난수)**를 사용하여 신선성(freshness)과 재전송 공격 방지를 보장한다.

프로토콜 핵심 흐름

- **Message 1 : A → B**
  - A는 nonce N_a를 생성하고,
  - N_a와 자신의 이름 A를 **B의 공개키로 암호화**하여 B에게 전송.
  - 목적: 오직 B만이 메시지를 복호화할 수 있음을 보장하여 “진짜 B와 통신 중임”을 확인.
- **Message 2 : B → A**
  - B는 새 nonce N_b 생성.
  - N_a와 N_b를 **A의 공개키로 암호화**하여 A에게 전송.
  - 목적: B가 A에게서 받은 N_a를 되돌려줌으로써 “자신이 진짜 B임”을 A에게 증명.
- **Message 3 : A → B**
  - A는 받은 N_b를 **B의 공개키로 암호화**해 다시 전송.
  - 목적: A가 실제로 메시지 2를 복호화할 수 있었음을 B에게 증명.

- 목적 : “A만이 B가 보낸 Message 2를 복호화할 수 있었고, 따라서 지금 메시지를 보내는 주체가 진짜 A이다” 라는 사실을 **B에게 증명**

프로토콜의 결과

프로토콜 종료 시:

- **A는 자신이 B와 통신하고 있음을 확신**한다.
- **B도 자신이 A와 통신하고 있음을 확신**한다.

양측 모두 상대방의 신원을 검증하는 데 성공하여 안전한 세션을 수립할 수 있다.

핵심 포인트 요약

- NSPK는 공개키 암호와 nonce를 이용한 **상호 인증 프로토콜**이다.
- 각 메시지는 상대의 공개키로 암호화되어, 오직 해당 주체만 복호화 가능하다.
- 최종적으로 A와 B는 서로를 신뢰할 수 있는 인증된 세션을 수립한다.

14.3 Modeling NSPK in Maude

- **목표:** Needham–Schroeder Public-Key(NSPK) 프로토콜을 **Maude**로 형식적으로 모델링하는 방법을 보여주는 섹션.
- 비공식 설명은 **프로토콜 1회 실행(run)**만 이야기하지만,
- Maude 모델은:
  - 여러 에이전트,
  - 여러 동시 세션(sessions),
  - Initiator/Responder 역할을 모두 고려하는 **좀 더 현실적인 모델**을 만든다.

전체 모델에 대한 전제/가정

1. 에이전트와 세션
   - 시스템에는 **2명 이상**의 에이전트가 있을 수 있다.
   - 프로토콜은 **여러 개의 동시 세션**이 존재할 수 있다.
   - 한 에이전트는
     - 어떤 실행(run)에서는 **initiator**,
     - 다른 실행에서는 **responder**,
     - 혹은 두 역할을 모두 할 수 있다.
   - 단순화를 위해 다음을 가정:
     - 에이전트 A는 **같은 responder와 동시에 두 개 이상의 run을 시작하지 않는다.**
     - 하지만 **두 에이전트가 서로를 향해 동시에 세션을 시작하는 것**은 허용된다.
2. 공격자/암호 관련 기본 가정 (Dolev–Yao 스타일)

   모든 에이전트(공격자 포함)에 대해 다음을 가정한다.

   1. **Nonce/비밀키 추측 불가**

- 모르는 nonce 값이나 비밀키 값을 성공적으로 추측할 수 없다.

2. **복호화 키 없이 복호화 불가**
   - 복호화 키를 모르는 암호문(ciphertext)을 복호화할 수 없다.
3. **암호화 키 없이 암호화 불가**
   - 암호화에 사용되는 키 값을 모르면 평문을 암호화할 수 없다.

이 가정 덕분에 실제 수학적 암호 알고리즘 대신, "키를 알고 있다/모른다" 여부만으로 모델링할 수 있다.

Nonce와 Key 모델링

1. Nonce
   - **sort 정의**

     ```
     sort Nonce .
     op nonce : Oid Nat -> Nonce [ctor] .
     ```

   - 의미:
     - `nonce(A, i)` : "에이전트 A가 만든 i번째 nonce"
     - 실제 숫자 값은 중요하지 않고, **(에이전트, 순번)** 쌍으로 추상화한다.
2. Public Key
   - **sort 정의**

     ```
     sort Key .
     op pubKey : Oid -> Key [ctor] .
     ```

   - 의미:
     - `pubKey(A)` : 에이전트 A의 공개키.
   - **Private key는 모델링하지 않음.**
     - 이유: "A의 공개키로 암호화된 것은 A만 풀 수 있다"는 가정을 사용하기 때문

       → 비밀키를 따로 term으로 나타낼 필요 없음.

메시지 구조 모델링

1. NSPK 메시지 일반 형태

   NSPK의 세 메시지는 모두 이런 형태를 갖는다:
   - `O1 . O2 . { message-content }_K`
     - `O1`, `O2`: 에이전트 식별자(보내는 사람, 받는 사람)
     - `message-content`: nonce/에이전트 이름 등을 포함
     - `_K`: 어떤 키로 암호화했는지
2. 평문 메시지 내용(PlainTextMsgContent)

   ```
   sorts PlainTextMsgContent EncrMsgContent .
   ```

```
op _;_ : Nonce Oid   → PlainTextMsgContent [ctor] . --- Message 1
op _;_ : Nonce Nonce → PlainTextMsgContent [ctor] . --- Message 2
subsort Nonce < PlainTextMsgContent .              --- Message 3
```

- 세 가지 경우를 지원:
  1. `Nonce ; Oid` → Message 1 (예: `Na ; A`)
  2. `Nonce ; Nonce` → Message 2 (예: `Na ; Nb`)
  3. `Nonce` 자체 → Message 3
- 여기서 `;` 는 메시지 요소를 붙이는 **연결 연산자** (본문에서의 `.` 대신 사용).

3. 암호화 연산

```
op encrypt_with_ : PlainTextMsgContent Key → EncrMsgContent [ctor] .
subsort EncrMsgContent < MsgContent .
```

- `encrypt M with K`

  → 평문 내용 `M` 을 키 `K` 로 암호화한 암호문.
- `EncrMsgContent` 는 `MsgContent` 의 하위 sort

  → "암호문도 메시지 내용의 한 종류"로, 네트워크 위에 돌아다니는 message wrapper의 content로 사용된다.

4. 메시지 예시 (Message 1)

```
msg (encrypt (nonce(A,3) ; A) with pubKey(B)) from A to B .
```

- 의미:
  - A가 B의 공개키로 `nonce(A,3) ; A` 를 암호화하여 B에게 보낸 Message 1.

Initiator 모델링

1. Initiator 클래스

```
class Initiator | initSessions : InitSessions, nonceCtr : Nat .
```

- `initSessions` :
  - Initiator가 현재 **어떤 상대와 어떤 상태의 세션을 가지고 있는지** 저장.
- `nonceCtr` :
  - A가 다음에 만들 nonce의 인덱스(번호).
  - `nonce(A, N)` 만들고 나면 `N` 을 1 증가.

2. InitSessions / Sessions 타입

```
sorts Sessions InitSessions .
subsort Sessions < InitSessions .

op emptySession : → Sessions [ctor] .
```

```
op __ : InitSessions InitSessions -> InitSessions
   [ctor assoc comm id: emptySession] .

op __ : Sessions Sessions -> Sessions
   [ctor assoc comm id: emptySession] .

op notInitiated     : Oid       -> InitSessions [ctor] .
op initiated        : Oid Nonce -> InitSessions [ctor] .
op trustedConnection : Oid      -> Sessions     [ctor] .
```

- `emptySession` : 아무 세션도 없는 상태.
- `__` : multiset 덧셈처럼 여러 세션 상태를 합치는 연산 (결합, 교환, 항등원 emptySession).
- 상태 표현:
  - `notInitiated(B)`

    → "A가 B와 세션을 시작하고 싶지만 아직 Message 1 안 보냄"
  - `initiated(B, N)`

    → "A가 B에게 nonce N으로 Message 1을 보냈고 Message 2를 기다리는 중"
  - `trustedConnection(B)`

    → "A는 B와 인증된 연결을 맺었다고 믿음"

3. 변수 선언

```
vars A B : Oid .
vars NONCE NONCE' : Nonce .
vars M N : Nat .
var IS : InitSessions .
```

- A, B: 에이전트
- NONCE, NONCE': nonce 값
- N: nonceCtr 값
- IS: 기존 initSessions 내용

규칙 send-1 (Message 1 전송)

- **역할:** Initiator A가 B에게 NSPK의 Message 1을 보내는 동작을 모델링.
- 조건 (왼쪽 상태)

```
< A : Initiator | initSessions : notInitiated(B) IS,
                  nonceCtr : N >
```

  - A의 상태:
    - B에 대한 세션 상태: `notInitiated(B)`

      → B와 통신을 시작하고 싶다.

    - nonceCtr = N
- 결과 (오른쪽 상태 + 보낸 메시지)

```
< A : Initiator | initSessions : initiated(B, nonce(A,N)) IS,
                  nonceCtr : N + 1 >
msg (encrypt (nonce(A,N) ; A) with pubKey(B)) from A to B .
```

  - A의 내부 상태 변화:
    - B에 대한 세션 상태: `initiated(B, nonce(A,N))`
    - nonceCtr: N+1로 증가
  - 네트워크로 나가는 메시지:
    - `encrypt (nonce(A,N) ; A) with pubKey(B)`

      → NSPK Message 1

규칙 read-2-send-3 (Message 2 수신 후 Message 3 전송)

- **역할:** A가 B로부터 NSPK Message 2를 받고, 검증 후 Message 3을 보내는 단계.
- 패턴 (수신된 메시지 + A의 현재 상태)

```
(msg (encrypt (NONCE ; NONCE') with pubKey(A)) from B to A)
< A : Initiator | initSessions : initiated(B, NONCE) IS >
```

  - B → A 메시지:
    - `(NONCE ; NONCE')`를 A의 공개키로 암호화한 것
    - NSPK Message 2에 해당 (`Na ; Nb`)
  - A의 상태:
    - B에 대해 `initiated(B, NONCE)` 상태
    - 즉, 이전에 B에게 NONCE를 보내두었음.
- 결과 상태

```
< A : Initiator | initSessions : trustedConnection(B) IS >
msg (encrypt NONCE' with pubKey(B)) from A to B .
```

  - A는:
    - 수신한 첫 번째 nonce가 자신이 저장한 NONCE와 같으므로,

      B와 인증된 연결을 맺었다고 판단 → `trustedConnection(B)`로 변경.
  - A는:
    - B의 nonce NONCE'을 B의 공개키로 암호화하여 B에게 Message 3 전송.

Responder 모델링

1. Responder 클래스

```
class Responder | respSessions : RespSessions, nonceCtr : Nat .
```

- `respSessions` :
  - 해당 에이전트가 **responder로 참여 중인 세션의 상태**를 저장.
- `nonceCtr` :
  - Responder가 새 nonce를 만들 때 쓰는 카운터.

2. RespSessions 타입

```
sort RespSessions .
subsort Sessions < RespSessions .

op __ : RespSessions RespSessions -> RespSessions
   [ctor assoc comm id: emptySession] .

op responded : Oid Nonce -> RespSessions [ctor] .
```

- `responded(A, N)` :
  - "B가 A의 Message 1을 받았고, 자신의 nonce N으로 응답했다"는 정보.

(본문 어딘가에 `trustedConnection(A)` 도 Sessions 쪽에 재사용됨.)

규칙 read-1-send-2 (Message 1 수신 후 Message 2 송신)

- **역할:** Responder B가 A의 Message 1을 받고, 자신만의 nonce를 만들어 Message 2를 보내는 단계.
- 조건

```
var RS : RespSessions .

crl [read-1-send-2] :
 (msg (encrypt (NONCE ; A) with pubKey(B)) from A to B)
 < B : Responder | respSessions : RS, nonceCtr : N >
 =>
 ...
 if not A inSession RS .
```

  - A → B 메시지:
    - `(NONCE ; A)` 를 B의 공개키로 암호화한 것 (Message 1)
  - B 상태:
    - respSessions = RS, nonceCtr = N
  - 조건 `not A inSession RS` :
    - B가 이미 A와 세션을 진행 중이면 안 됨 (중복 세션 방지).
- 결과

```
< B : Responder | respSessions : responded(A, nonce(B,N)) RS,
                  nonceCtr : N + 1 >
msg (encrypt (NONCE ; nonce(B,N)) with pubKey(A)) from B to A
```

  - B는:
    - 새 nonce `nonce(B,N)` 생성
    - `responded(A, nonce(B,N))`를 respSessions에 기록
    - nonceCtr를 N+1로 증가
  - 메시지:
    - `(NONCE ; nonce(B,N))`를 A의 공개키로 암호화하여 A에게 전송

      → NSPK Message 2

규칙 read-3 (Message 3 수신)

- **역할:** B가 A의 Message 3을 받고, 연결이 인증되었음을 확인하는 단계.
- 패턴

```
rl [read-3] :
  (msg (encrypt NONCE with pubKey(B)) from A to B)
  < B : Responder | respSessions : responded(A, NONCE) RS >
  ⇒
  < B : Responder | respSessions : trustedConnection(A) RS > .
```

  - A → B 메시지:
    - NONCE를 B의 공개키로 암호화한 Message 3
  - B 상태:
    - 이전에 `responded(A, NONCE)`가 기록되어 있음

      (자신이 그 세션에서 NONCE를 A에게 준 상태)
- 결과
  - B는 NONCE가 예상한 값과 같음을 확인하고, 세션 상태를 `trustedConnection(A)`로 변경

    → "A와 인증된 연결이 완료되었다"고 판단.

Initiator이자 Responder인 에이전트

마지막으로, 한 에이전트가 **Initiator와 Responder 역할을 모두 할 수 있는 경우**를 모델링한다.

```
class InitAndResp .
subclass InitAndResp < Initiator Responder .
endom)
```

- `InitAndResp` 클래스는
  - `Initiator`와 `Responder`의 **서브클래스**.
- 따라서:

- 두 클래스의 속성(`initSessions`, `respSessions`, `nonceCtr`)을 모두 가지며,
- 두 클래스의 rewrite 규칙(initiator 규칙 + responder 규칙)을 모두 상속.

- 이런 에이전트는 한 세션에서는 initiator, 다른 세션에서는 responder로 동작할 수 있다.

전체 흐름 한눈에 요약

1. **Nonces/Keys/메시지 구조**를 추상적인 term으로 정의한다.
2. **Initiator**는:
   - B와 시작 전: `notInitiated(B)`
   - Message 1 보냄: `initiated(B, Na)`
   - Message 2 확인 + Message 3 보냄: `trustedConnection(B)`
3. **Responder**는:
   - Message 1 받음 + Message 2 보냄: `responded(A, Nb)`
   - Message 3에서 Nb 확인: `trustedConnection(A)`
4. 이 모든 과정이 Maude의 **rewrite 규칙(send-1, read-2-send-3, read-1-send-2, read-3)** 로 표현된다.
5. `InitAndResp`는 Initiator와 Responder 둘 다 가능한 에이전트 타입.

14.3.1 Executing the NSPK Specification

실험 셋업 (init2)

- 에이전트 세 개:
  - `"a"` : InitAndResp (initiator+responder)
  - `"Bank"` : Responder
  - `"c"` : InitAndResp
- 초기 상태 `init2`에서:
  - `"a"`는 **"c"와만** 세션을 시작하고 싶어 함 → `notInitiated("c")`
  - `"c"`는 `"Bank"`와 `"a"` 둘 다와 세션을 시작하고 싶어 함
    → `notInitiated("Bank")`, `notInitiated("a")`
  - `"Bank"`는 responder로만 있음, 아직 아무 세션도 없음.
- 의도:
  - **"a"와 "Bank"는 서로 신뢰 연결을 맺으면 안 된다.ㄴㅁ**
    (a는 Bank와 통신하고 싶지 않다는 설정)

Maude 검색 명령

```
(search init2 =>! C:Configuration .)
```

- 의미:
  - 초기 상태 `init2`에서 출발해

- 도달 가능한 **모든 최종 상태(final state)** `C` 를 찾아라.

- `=>!` : 더 이상 rewrite가 불가능한 상태(막다른 상태 = final state)를 의미.

검색 결과 (유일한 final state)

- 딱 **한 개의 solution**만 나옴.
- 그 상태에서의 관계:
  - `"Bank"`
    - `respSessions : trustedConnection("c")`

      → Bank는 **"c"와만** 신뢰 연결을 맺음.
  - `"a"`
    - `initSessions : trustedConnection("c")`
    - `respSessions : trustedConnection("c")`

      → a와 c는 서로 **양방향으로** trustedConnection.
  - `"c"`
    - `initSessions : trustedConnection("Bank"), trustedConnection("a")`
    - `respSessions : trustedConnection("a")`

      → c는 Bank와 a 모두와 신뢰 연결.
- **중요 포인트:**
  - 어떤 에이전트도 `"a"` – `"Bank"` 사이에 `trustedConnection`을 갖고 있지 않음

    → **a와 Bank는 끝까지 서로 인증된 연결을 만들지 않음.**

결론

- 나쁜 공격자("bad guys")가 없는 모델에서 NSPK를 실행해 보니:
  - **원하던 연결들(a–c, c–Bank)은 잘 성립**하고,
  - **원하지 않던 연결(a–Bank)은 절대 생기지 않는다.**
- 따라서 이 설정에서는 "공격자가 없을 때 NSPK는 의도한 대로 정상 동작한다"라는 결론을 내린다.

14.4 Modeling Intruders

침입자 모델의 목적과 가정

- 왜 침입자를 모델링하나?
  - 지금까지는 **"나쁜 놈이 없는"** 상태에서 NSPK 프로토콜이 잘 동작하는지만 봤음.
  - 실제 네트워크에서는 공격자가 있으니, **공격자가 할 수 있는 모든 행동을 형식적으로 모델링**하고, 그 안에서 NSPK가 여전히 안전한지 확인하려는 것.
- 사용한 모델: Dolev–Yao Intruder

  침입자(intruder)는 Dolev–Yao 모델을 따른다고 가정:
  - 네트워크 상의 메시지를

        - **도청(overhear) / 가로채기(intercept)** 가능
    - 자신의 **공개키**로 암호화된 메시지는 **복호화** 가능
    - 알고 있는 **nonce**를 이용해 **새 메시지를 만들어 주입** 가능
    - 지금까지 본 **어떤 메시지든 재전송(replay)** 가능 (암호문을 이해하지 못해도 재전송할 수 있고, 평문 부분은 수정할 수도 있음)
    - 공격자도 네트워크의 한 노드로서 **정상 프로토콜 run에 참여**할 수 있고, 사용 중인 프로토콜도 알고 있다고 가정.
- 아이디어: "침입자가 할 수 있는 것은 네트워크 상에서 보낸/받은 메시지를 '조합 • 암호화 • 복호화 • 재전송'하는 것뿐"이라고 보고, 이 모든 경우를 시스템 안에 rewrite rule로 때려 넣는다.

Intruder 객체 구조

- Maude 모듈 선언

```
omod NSPK-INTRUDER is
  including NSPK .
  including OID-SET .
```

    - NSPK 프로토콜 모듈과 OID(에이전트 ID 집합) 모듈을 포함하는 새 모듈.
- 사용 변수들 (intruder 규칙에서 쓰는 변수들)
    - `NONCE, NONCE' : Nonce` – 논스
    - `NSET : NonceSet` – 논스들의 집합
    - `ENCRMSG : EncrMsgContent`, `ENCRMSGS : EncrMsgContentSet` – 암호문 내용과 그 집합
    - `N : Nat` – 자연수 (nonce 카운터 등)
    - `MSGC : PlainTextMsgContent` – 평문 메시지 내용
    - `A B I O O' O'' : Oid` – 에이전트 ID들 (sender, receiver, intruder 등)
    - `OS : OidSet` – 에이전트 ID 집합
    - `IS : InitSessions`, `RS : RespSessions` – initiator / responder 세션 정보
- Intruder 클래스 정의

```
class Intruder |
  initSessions : InitSessions,
  respSessions : RespSessions,
  nonceCtr : Nat,
  agentsSeen : OidSet,
  noncesSeen : NonceSet,
  encrMsgsSeen : EncrMsgContentSet .
```

    - **일반 에이전트와 동일한 속성**
        - `initSessions`, `respSessions`, `nonceCtr` : NSPK에서 initiator / responder로 동작할 때 필요한 정보
    - **침입자 고유의 지식(knowledge) 저장 속성**

- `agentsSeen` : 지금까지 **알게 된 에이전트 ID**들의 집합
- `noncesSeen` : 지금까지 **알게 된 nonce**들의 집합
- `encrMsgsSeen` : **복호화는 못 했지만 본 적 있는 암호문 메시지 내용**들의 집합

보조 자료형: NonceSet, EncrMsgContentSet

- NonceSet

```
sort NonceSet .
subsort Nonce < NonceSet .
op emptyNonceSet : -> NonceSet [ctor] .
op __ : NonceSet NonceSet -> NonceSet
      [ctor assoc comm id: emptyNonceSet] .
eq NONCE NONCE = NONCE .
```

  - `NonceSet`은 논스들의 **집합**을 표현.
  - `__` 연산자는 집합 합집합처럼 사용 (associative, commutative, identity).
  - `eq NONCE NONCE = NONCE`로 **중복 제거**.
- EncrMsgContentSet
  - `EncrMsgContentSet`도 `NonceSet`과 같은 형식으로 정의 (암호문 메시지 내용들의 집합).

"정상 프로토콜 행동"으로서의 침입자 규칙들

- 침입자가 **정상 에이전트처럼** NSPK 프로토콜에 참여하는 규칙이 4개 있음:
  - Initiator 역할 규칙 2개
  - Responder 역할 규칙 2개 (책에서 그 중 Message 1 수신 규칙만 자세히 보여주고, 나머지는 연습문제로 넘김)
- Message 1 수신 + 응답 규칙

```
crl [intruder-receive-message-1] :
  (msg (encrypt (NONCE ; A) with pubKey(I)) from A to I)
  < I : Intruder | respSessions : RS,
                   nonceCtr : N,
                   agentsSeen : OS,
                   noncesSeen : NSET >
=>
  < I : Intruder | respSessions :
                     responded(A, nonce(I,N)) RS,
                   nonceCtr : N + 1,
                   agentsSeen : OS ; A,
                   noncesSeen : NSET NONCE nonce(I,N) >
  msg (encrypt (NONCE ; nonce(I,N)) with pubKey(A)) from I to A
  if not A inSession RS .
```

  해석:

  - A가 침입자 I에게 NSPK **Message 1**(`{Na, A}_PK(I)`)을 보냈을 때:

1. I는 **정상 NSPK responder처럼** fresh nonce `nonce(I,N)`을 만들고,
2. `responded(A, nonce(I,N))`로 세션 상태를 기록하고,
3. `agentsSeen`에 A를 추가,
4. `noncesSeen`에 받은 `NONCE`(Na)와 새로 만든 `nonce(I,N)`를 추가,
5. Message 2 형식의 응답

   `{Na, nonce(I,N)}_PK(A)` 를 A에게 보냄.

- 조건 `not A inSession RS` : 이미 I가 A와 responder 세션을 진행 중이면 또 시작하지 않도록.

- 암호문을 훔치되 이해는 못 하는 규칙

```
crl [intercept-but-not-understand] :
  (msg ENCRMSG from O' to O)
  < I : Intruder | agentsSeen : OS,
                   encrMsgsSeen : ENCRMSGS >
=>
  < I : Intruder | agentsSeen : OS ; O ; O',
                   encrMsgsSeen : ENCRMSG ENCRMSGS >
  if O =/= I .
```

  - I가 **자기에게 온 게 아닌** 암호문 메시지를 가로채는 상황.
  - 암호문은 해독 못하지만,
    - 발신자 `O'`,
    - 수신자 `O`,
    - 암호문 `ENCRMSG`

      를 자신의 지식에 기록.

- overhear + 이해하는 규칙들 (msg1/msg2)
  - 책에 나온 두 규칙:

```
rl [intercept-msg1-and-understand] :
  (msg (encrypt (NONCE ; A) with pubKey(I)) from O to I)
  < I : Intruder | agentsSeen : OS,
                   noncesSeen : NSET >
=>
  < I : Intruder | agentsSeen : OS ; O ; A,
                   noncesSeen : NSET NONCE > .

rl [intercept-msg2-and-understand] :
  (msg (encrypt (NONCE ; NONCE') with pubKey(I)) from O to I)
  < I : Intruder | agentsSeen : OS,
                   noncesSeen : NSET >
=>
  < I : Intruder | agentsSeen : OS ; O,
                   noncesSeen : NSET NONCE NONCE' > .
```

- I에게 오면서 **I의 공개키로 암호화된 메시지**는 I가 복호화 가능.
- msg1에선 nonce 하나, msg2에선 nonce 두 개를 꺼내서 `noncesSeen`에 저장.
- `agentsSeen`에는 발신자와 (msg1일 때) 포함된 A를 추가.

- 단순 수신 후 폐기하는 규칙들
  - 침입자가 메시지를 받지만, 정보만 추출하고 **메시지 자체는 버리는** 경우.
  - 이유:
    - 다른 침입자가 보낸 가짜일 수 있음.
    - 더 이상 NSPK 정상 프로토콜을 이어가고 싶지 않을 수 있음 (목표가 nonce 수집이면 충분).

    (해당 세 개 규칙은 이미지에 다 안 나왔지만, 역할은 "정보만 저장 후 메시지는 discard" 정도로 이해하면 됨.)

침입자의 가짜 메시지 생성 능력

이제부터는 침입자가 **공격자로서 행동**하는 규칙들.

- 저장된 암호문을 그대로 재사용 – send-encrypted

```
crl [send-encrypted] :
  < I : Intruder | encrMsgsSeen :
      (encrypt MSGC with pubKey(B)) ENCRMSGS,
      agentsSeen : A ; OS >
⇒
  < I : Intruder >
  (msg (encrypt MSGC with pubKey(B)) from A to B)
  if A =/= B .
```

  - I가 예전에 가로챈 암호문 `{MSGC}_PK(B)`를 알고 있을 때,
  - 그걸 **임의의 발신자 A**(침입자가 알고 있는 에이전트)에서 B로 가는 메시지로 재전송.
  - 이때 내용은 그대로지만, "보낸 사람"이 바뀌므로 **위조(sender spoofing)**.

  독자 의문: I가 이 암호문이 PK(B)로 암호화된 것임을 어떻게 아는가? → 가로챈 시점에 메시지의 수신자를 봤기 때문에 안다고 가정.

- Message 1 형식의 가짜 메시지 – send-1-fake

```
crl [send-1-fake] :
  < I : Intruder | agentsSeen : A ; B ; OS,
                   noncesSeen : NONCE NSET >
⇒
  < I : Intruder >
  (msg (encrypt (NONCE ; A) with pubKey(B)) from A to B)
  if A =/= B /\ B =/= I .
```

  - 침입자가
    - 어떤 에이전트 A, B를 알고 있고,
    - nonce 하나 `NONCE`를 알고 있을 때,

- NSPK **Message 1** 꼴인 `{NONCE, A}_PK(B)`를 위조해서 A가 B에게 보낸 것처럼 전송.

- Message 2 형식의 가짜 메시지 – send-2-fake

```
crl [send-2-fake] :
  < I : Intruder | agentsSeen : A ; B ; OS,
                   noncesSeen : NONCE NONCE' NSET >
=>
  < I : Intruder >
  (msg (encrypt (NONCE ; NONCE') with pubKey(A)) from B to A)
  if A =/= B /\ A =/= I /\ B =/= I .
```

  - 침입자가
    - A, B의 ID를 알고 있고,
    - nonce 두 개 `NONCE`, `NONCE'`를 알고 있으면,
  - NSPK **Message 2** 꼴 `{NONCE, NONCE'}_PK(A)`를 위조해서 B가 A에게 보낸 것처럼 전송.

  이 세 규칙들이 실제로 NSPK 공격(예: Lowe 공격)을 구성할 때 핵심 역할을 한다.

중복 메시지 제거를 통한 상태 공간 축소

침입자는 같은 가짜 메시지를 여러 번 보낼 수 있음 → 상태 안에 **동일한 메시지의 복사본이 여러 개** 생길 수 있다.

하지만:

- 어떤 상태 S에 메시지 M이 여러 개 있더라도,
- M을 하나만 남기고 제거한 상태에서도 **동일한 행동이 모두 가능**하다.

그래서 굳이 복사본을 유지할 필요가 없으므로, 다음 등식으로 **중복 메시지를 자동 제거**한다.

```
var MSG : Msg .
eq MSG MSG = MSG .
```

- `MSG MSG` → `MSG` : 같은 메시지가 두 개 붙어 있으면 하나로 축소.
- 이를 통해 **상태 공간(state space)을 줄여** 탐색 효율을 높인다.

14.5 Analyzing NSPK with Intruders

이 절은 Maude 모델을 이용해 **NSPK(Needham–Schroeder Public-Key)** 프로토콜이 공격 가능한지 분석한다.

여기서 Beagle Boys(BB)는 침입자(공격자)로 설정되어 있으며, 그 목적은 은행(Bank)이 공격자인 Beagle Boys를 Scrooge라고 잘못 믿고 인증된 연결이 성립되었다고 착각하도록 만드는 것이다.

Scrooge는 원래 **은행과는 통신하기를 원하지 않고**, Beagle Boys와만 소통하려 한다.

Maude의 탐색 결과 다음이 확인된다:

- 실제로 은행은 자신이 Scrooge와 인증된 연결을 맺었다고 믿게 되는 상태에 도달할 수 있으며, 이때 실제 메시지는 **Beagle Boys가 중간에서 가로채고 재전송하여 만들어낸 것**이다.

즉,

- 은행은 "Scrooge와 인증이 완료됨"이라고 착각함
- 하지만 그 과정은 Beagle Boys가 만든 가짜 흐름
- **NSPK 프로토콜은 중간자 공격(MITM)에 취약하다**

따라서 NSPK는 실제로 공격 가능한 프로토콜임이 증명된다.

공격 시나리오 요약

1. 초기 상태
   - Scrooge는 Beagle Boys와 통신을 시도함 (은행과 통신하고 싶지 않음)
   - Beagle Boys는 공격자 역할
   - 은행은 정상적인 수신자
2. 전체 공격 흐름 (간단 요약)
   a. **Scrooge → BB : 메시지 1 전송**

   Scrooge는 Beagle Boys에게 Message 1을 보냄 → Beagle Boys는 이 메시지를 이용해 **Scrooge인 척 가장**함

   b. **BB(가짜 Scrooge) → 은행 : 메시지 1 전달**

   은행은 이를 Scrooge의 요청이라 믿고 Message 2를 Scrooge로 보냄

   c. **은행 → Scrooge : 메시지 2**

   Beagle Boys는 이 메시지를 **훔쳐보지만 암호를 풀 수는 없음** → 대신 저장해두고 나중에 **그대로 Scrooge에게 재전송(replay)**

   d. **Scrooge → BB : 메시지 3**

   Scrooge는 본인이 BB와 통신 중이라 생각하고 은행이 보낸 nonce(Nb)를 **Beagle Boys의 공개키로 암호화하여 돌려보냄**

   이 과정에서 Scrooge는 **은행의 nonce Nb**를 Beagle Boys에게 넘겨버림.

   e. **BB → 은행 : 메시지 3 위조**

   Beagle Boys는 Nb를 알게 되었으므로 은행이 기다리고 있는 Message 3을 **Scrooge인 척 위조해 은행에 보냄**
3. **결과: 공격 성공**

   은행은 Scrooge와 인증되었다고 착각 → Beagle Boys는 Scrooge 계좌의 돈을 마음대로 이동 가능

왜 공격이 가능한가?

NSPK의 구조상 **중간자가 메시지를 재전송(replay)하여 인증을 속일 수 있는 취약점**이 존재한다.

Beagle Boys는

- 자신이 받은 메시지를 복호화할 수 없어도
- **그대로 재전송하는 것만으로도** 프로토콜을 속일 수 있다.

공식적인 메시지 흐름

```
S1.M1: S → BB      : S . BB . {Ns, S}PK_BB
S2.M1: BB(S) → B   : S . B  . {Ns, S}PK_B
```

```
S2.M2: B → BB(S)  : B . S  . {Ns, Nb}PK_S
S1.M2: BB → S     : BB . S . {Ns, Nb}PK_S
S1.M3: S → BB     : S . BB . {Nb}PK_BB
S2.M3: BB(S) → B  : S . B  . {Nb}PK_B
```

모든 단계는 프로토콜 규칙을 위반하지 않으며, **은행과 Scrooge 모두 정상적으로 프로토콜을 따르고 있음에도 공격이 성립**한다.

결론

- Beagle Boys(침입자)는 **중간자 공격(MitM) + Replay Attack**으로 NSPK를 뚫을 수 있다.
- 은행은 Scrooge와 인증이 확립되었다고 착각하게 된다.
- 이는 **NSPK 프로토콜이 근본적으로 안전하지 않음을 의미**한다.
- 이 취약점은 Lowe 공격으로 알려진 유명한 NSPK 결함이다.

14.6 Discussion

- **NSPK 프로토콜은 오랫동안 안전한 것으로 믿어졌지만 실제로는 취약하다.**
  - NSPK는 **1978년에 발표**된 이후, **1996년의 유명한 암호학 서적에서도 안전하지 않다는 언급이 없었고**, **1989년에는(침입자가 없다는 가정 하에) 안전하다는 증명도 있었다.**
- **그러나 1995년 Gavin Lowe가 실제 공격을 발견했다.**
  - Lowe는 CSP용 FDR 도구를 이용한 *형식 분석(formal analysis)* 중에 **대학자들도 17년 동안 놓쳤던 공격 시나리오를 찾아냄.**
  - Maude로 분석한 공격도 Lowe가 발견한 공격과 완전히 동일하다.
- **Maude 탐색이 오래 걸린 이유**
  - 문제 자체가 복잡하기 때문이며, 실제로 Lowe가 찾을 때까지 이 공격은 17년 동안 발견되지 않았다.
  - 원래 프로토콜 실행(run)에 참여하는 규칙들을 제외하면 **Maude는 몇 초 만에 공격을 찾을 수 있다.**
  - 또는 "공격자가 특정 nonce/키를 이미 알고 있다"는 조건으로 탐색하면 **약 15초만에 공격 상태를 찾을 수 있다.**
- **Lowe의 연구는 암호 프로토콜 분석 방식에 큰 영향을 주었다**
  - 사람 손으로는 이런 취약점을 검증하기가 점점 불가능해짐을 보여줌.
  - 이후 많은 자동화된 형식 검증 도구들이 개발됨:
    - TAMARIN Prover
    - Scyther
    - ProVerif
    - AVISPA
    - Maude 기반 Maude-NPA

14.7 The Corrected Protocol

Lowe는 NSPK 프로토콜의 공격을 막기 위해 Message 2 안에 **응답자 B의 ID**를 넣는 방식으로 수정했다.

$$\text{Message 2.} \qquad B \rightarrow A \ : \qquad B \,.\, A \,.\, \{N_a \,.\, N_b \,.\, B\}_{PK_A}$$

이 수정된 프로토콜에 대해:

- 검색(search)은 오류를 찾는 방법일 뿐, 공격이 없다고 해서 절대 안전함을 "증명"하는 것은 아니다.
- 하지만 Lowe는 **만약 이 수정된 프로토콜이 깨질 수 있다면**, 단 **Initiator 1명 + Responder 1명 + Intruder 1명** 만으로도 공격이 가능함을 보였다.
- 각 에이전트는 **nonce 하나만** 생성하면 되므로, 상태 공간은 **유한**하다.
- 따라서 검색은 **반드시 끝나며**, 세 에이전트 상태에서 공격이 없다면 **프로토콜은 안전하다는 것을 증명할 수 있다.**

즉, **수정된 NSPK 프로토콜은 자동 검증이 가능한 수준으로 단순하며, 검색만으로 안전성이 증명되는 구조이다.**