University of Camerino

School of Science and Technology

Master Degree in Computer Science (LM-18)

Modeling and Verification of the Needham–Schroeder Protocol

Student

Giorgio Saldana

A.A. 2023/2024

Contents

Abstract

This report details the modeling and verification of the Needham-Schroeder protocol, a key authentication protocol, using SPIN and UPPAAL model checkers. The protocol, despite its widespread use, is susceptible to a man-in-the-middle (MITM) attack, which compromises its security. The study begins by modeling the vulnerable version of the protocol in SPIN to identify the MITM vulnerability. The model checker successfully detects this security flaw, underscoring the protocol's weakness. Hence wil be modelled a corrected version of the Needham-Schroeder protocol in SPIN, demonstrating that the security flaw can be mitigated. This corrected version undergoes rigorous verification to ensure the MITM attack is no longer effective. Both the vulnerable and corrected versions are then modeled in UPPAAL, adapting the Linear Temporal Logic (LTL) properties used in SPIN to Computation Tree Logic (CTL) syntax suitable for UPPAAL. This adaptation involves a careful transformation of properties to fit UPPAAL's constraints while maintaining the integrity of the security checks.

The report provides a comprehensive documentation of the modeling process, the expression of security properties, and the results of the verification checks. Screenshots and detailed explanations illustrate the verification outcomes, offering valuable insights into the efficacy of SPIN and UPPAAL in identifying and addressing security vulnerabilities in cryptographic protocols. This study highlights the critical role of formal verification tools in enhancing the security of communication protocols. Tasks included documenting the protocol, modeling the vulnerable and corrected versions in SPIN, and adapting the models to UPPAAL, ensuring a non-deterministic, real-time-free verification process. This approach confirms the robustness of the corrected protocol against MITM attacks, contributing to the protocol's security analysis literature.

1. Introduction

The Needham-Schroeder protocol is a key authentication protocol used for secure communication between two parties over an insecure network. However, the protocol has known vulnerabilities, such as the man-in-the-middle attack, which can compromise its security. This report aims to model and verify both the vulnerable and corrected versions of the Needham-Schroeder protocol using two formal verification tools: SPIN and UPPAAL. By leveraging these tools, we can identify the presence of the vulnerability and verify the effectiveness of the corrected protocol. The objectives of this report are as follows:

1. To document the Needham-Schroeder protocol and its vulnerabilities.
2. To model the protocol in SPIN and use model checking to detect the man-in-the-middle vulnerability.
3. To model and verify the corrected version of the protocol in SPIN.
4. To adapt both versions of the protocol to the UPPAAL model checker and verify them using CTL properties.
5. To analyze and discuss the results of the verification processes.

The structure of the report is organized as follows:

- **Section 1** provides an overview of the Needham-Schroeder protocol and its vulnerabilities.
- **Section 2** details the modeling and verification process using SPIN.
- **Section 3** describes the adaptation and verification of the protocol in UPPAAL.
- **Section 4** presents the results and discussion.
- **Section 5** concludes the report with final observations and future work suggestions.

This comprehensive approach ensures a thorough analysis of the Needham-Schroeder protocol's security and demonstrates the utility of formal verification tools in protocol analysis.

2. The Needham-Schroeder Protocol and its Vulnerability

The Needham-Schroeder protocol consists of two versions: the Needham-Schroeder Symmetric Key Protocol and the Needham-Schroeder Public Key Protocol. Both protocols aim to provide mutual authentication between two parties over an insecure network. This document focuses on explaining both protocols, their vulnerabilities, and the corrections made to address these vulnerabilities.

2.1 Needham-Schroeder Symmetric Key Protocol

The Needham-Schroeder Symmetric Key Protocol involves a trusted third party, the Key Distribution Center (KDC), and is described as follows:

1. **A** $\rightarrow$ **S**: $\{A, B, N_A\}$
2. **S** $\rightarrow$ **A**: $\{K_{AB}, B, N_A, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$
3. **A** $\rightarrow$ **B**: $\{K_{AB}, A\}_{K_{BS}}$
4. **B** $\rightarrow$ **A**: $\{N_B\}_{K_{AB}}$
5. **A** $\rightarrow$ **B**: $\{f(N_B)\}_{K_{AB}}$

Where:

- **A** and **B** are the communicating parties.
- **S** is the trusted server.
- N_A and N_B are nonces generated by **A** and **B**, respectively.
- K_{AS} and K_{BS} are the symmetric keys shared between **A** and **S**, and **B** and **S**, respectively.
- K_{AB} is the session key generated by **S** for **A** and **B**.
- $\{M\}_K$ denotes the message M encrypted with key K.

2.2 Needham-Schroeder Public Key Protocol

The Needham-Schroeder Public Key Protocol uses public-key cryptography and is described as follows:

1. **A** $\rightarrow$ **B**: $\{N_A, A\}_{K_B}$
2. **B** $\rightarrow$ **A**: $\{N_A, N_B\}_{K_A}$
3. **A** $\rightarrow$ **B**: $\{N_B\}_{K_B}$

Where:

- **A** and **B** are the communicating parties.
- N_A and N_B are nonces generated by **A** and **B**, respectively.
- K_A and K_B are the public keys of **A** and **B**, respectively.
- $\{M\}_K$ denotes the message M encrypted with key K.

2.3 Man-in-the-Middle Attack

The Needham-Schroeder Public Key Protocol is vulnerable to a man-in-the-middle (MITM) attack, as discovered by Gavin Lowe in 1995. The attack proceeds as follows:

1. **A** $\rightarrow$ **I (intruder)**: $\{N_A, A\}_{K_I}$
2. **I** $\rightarrow$ **B**: $\{N_A, A\}_{K_B}$
3. **B** $\rightarrow$ **I**: $\{N_A, N_B\}_{K_A}$
4. **I** $\rightarrow$ **A**: $\{N_A, N_B\}_{K_A}$
5. **A** $\rightarrow$ **I**: $\{N_B\}_{K_I}$
6. **I** $\rightarrow$ **B**: $\{N_B\}_{K_B}$

In this attack, the intruder **I** intercepts messages between **A** and **B**, effectively posing as **A** to **B** and vice versa. This allows **I** to decrypt and re-encrypt messages, gaining unauthorized access to the information being exchanged.

2.4 Corrected Version

To mitigate this vulnerability, the corrected version of the protocol involves mutual authentication of nonces in the third step. The corrected steps are:

1. **A** $\rightarrow$ **B**: $\{N_A, A\}_{K_B}$
2. **B** $\rightarrow$ **A**: $\{B, N_A, N_B\}_{K_A}$
3. **A** $\rightarrow$ **B**: $\{N_B, A\}_{K_B}$

This modification introduces a nonce challenge-response mechanism in both directions, ensuring that each party verifies the received nonce with its corresponding identity. This process securely exchanges and validates the nonces, preventing MITM attacks.

3. SPIN

SPIN is a powerful model checker used for the formal verification of distributed systems. It uses Promela (Process Meta Language) to model system behaviors and properties. SPIN allows for the simulation and verification of system models to ensure correctness and detect potential errors such as deadlocks and security vulnerabilities.

3.1 Modeling the MITM Attack in Promela

To model the man-in-the-middle (MITM) attack on the Needham-Schroeder Public Key Protocol in Promela, we define the roles of the participants (Alice, Bob, and the Intruder), the messages they exchange and the communication channel

Listing 3.1: Promela code for Needham-Schroeder Protocol with MITM Attack

```
/* Enumeration of our symbolic constant to have a more readable
   model. */
mtype = {ok, err, msg1, msg2, msg3, keyA, keyB, agentA, agentB,
         nonceA, nonceB, agentI, keyI, nonceI };

/* Encrypted msg.  */
typedef Crypt { mtype key, content1, content2 };

/* Network modelled as a single message channel shared by agents. */
chan network = [0] of {mtype, /* msg number */
                       mtype, /* receiver */
                       Crypt
};

/* global variables for verification*/
mtype partnerA, partnerB;
mtype statusA = err;
mtype statusB = err;

bool knows_nonceA = false;
bool knows_nonceB = false;

/* Agent (A)lice */
active proctype Alice() {
        /* local variables */

        mtype pkey;      /* the other agent's public key
                              */
        mtype pnonce;    /* nonce that we receive from the other
            agent   */
```

```
Crypt messageAB; /* our encrypted message to the other party
        */
Crypt data;      /* received encrypted message
                     */


/* Initialization of non deterministic choice made by alice
   to choose the partner */

    if
:: partnerA = agentB; pkey = keyB
:: partnerA = agentI; pkey = keyI
fi ;


/* Prepare the first message */

messageAB.key = pkey;
messageAB.content1 = agentA;
messageAB.content2 = nonceA;

/* Send the first message to the other party */

network ! msg1 (partnerA, messageAB);

/* Wait for an answer. Observe that we are pattern-matching
   on the
messages that start with msg2 and agentA, that is, we block
   until
we see a message with values msg2 and agentA as the first
   and second
components. The third component is copied to the variable
   data. */

network ? (msg2, agentA, data);

/* We proceed only if the key field of the data matches keyA
    and the
received nonce is the one that we have sent earlier; block
otherwise.  */
if
:: (data.key == keyA) && (data.content1 == nonceA) ->

                /* Obtain Bob's nonce */

                pnonce = data.content2;

                /* Prepare the last message */
                messageAB.key = pkey;
                messageAB.content1 = pnonce;
                messageAB.content2 = 0;  /* content2 is not
                   used in the last message,just set it to 0
                    */


                /* Send the prepared messaage */
```

```
                        network ! msg3 (partnerA, messageAB);


                        /* and last - update the auxilary status
                           variable */
                        statusA = ok;

        fi ;

}

/* Agent (B)ob */
active proctype Bob() {
        /* local variables */

        mtype pkey;             /* other agent's pub-key */
        mtype pnonce;           /* nonce that we receive from the
           other agent   */
        Crypt messageAB;        /* our encrypted message to the
           other party     */
        Crypt data;             /* received encrypted message
                             */

        /* Initialization  */

        partnerB = agentA;
        pkey = keyA;



            /* Wait for the first interation from Alice taking a
               look that the message received respect the
               pattern-matching defined before. */

            network ? (msg1, agentB, data);

            /* Proceed if key field match with keyB  */
            if
            :: (data.key == keyB)  ->

                    /* Get Alice's nonce */
                pnonce = data.content2;

                /* Prepare the first message from Bob, so in the
                   protocol path scheme this is the 2nd msg sent.
                The key must be the Alice ones, the content1 must be
                    the nonce generated and sent from Alice,
                the content 2 must be the Bob's nonce
                */

                messageAB.key = pkey;
                messageAB.content1 = pnonce;
                messageAB.content2 = nonceB;

                /* Send the second message to the other parties */
                network ! msg2(partnerB, messageAB);
                /* Wait for the last message */

                network ? (msg3, agentB, data);

                /* We proceed only if the key field matches keyB and
                    the received nonce is the one we have sent
                   earlier */
                if
                :: (data.key == keyB) && (data.content1 == nonceB)
                   ->
                        /* Update the auxiliary status variable */
                        statusB = ok;

                fi ;

            fi ;

}

/* Agent (I)ntruder */

active proctype Intruder() {
    mtype msg, recpt;
    Crypt data, intercepted;
    do
        :: network ? (msg, _, data) ->

        if /* perhaps store the message */
        :: intercepted. key = data.key;
            intercepted.content1 = data.content1;
            intercepted.content2 = data.content2;
        :: skip;
        fi ;

        /* Check if intercepted message can be decrypted */
        if
        :: data.key = keyI ->
            if
            :: (data.content1 == nonceA || data.content2 == nonceA)
                -> knows_nonceA = true;
            fi ;
            if
            :: (data.content1 == nonceB) || (data.content2 == nonceB
                )
                -> knows_nonceB = true;
            fi ;
        fi ;

        :: /* Replay or send a message */
        if /* choose message type */
        :: msg = msg1;
        :: msg = msg2;
        :: msg = msg3;
        fi ;
        if /* choose a recepient */
        :: recpt = agentA;
        :: recpt = agentB;
        fi ;
        if /* replay intercepted message or assemble it */
            :: data.key      = intercepted.key;
               data.content1 = intercepted.content1;
               data.content2 = intercepted.content2;
            :: if /* assemble content1 */
               :: data.content1 = agentA;
               :: data.content1 = agentB;
               :: data.content1 = agentI;
               :: data.content1 = nonceI;
              fi ;
            if /* assemble key */
               :: data.key = keyA;
               :: data.key = keyB;
               :: data.key = keyI;
            fi ;

            data.content2 = nonceI;

        fi ;

        network ! msg (recpt, data);
    od
}
```

3.2 Agents in the Model

In the Promela model of the Needham-Schroeder protocol, we define three types of agents: Alice, Bob, and the Intruder.

3.2.1 Alice

Alice initiates the communication by sending a message to her chosen partner. She prepares the first message, sends it over the network, waits for a response, and finally verifies the received data to complete the protocol.

3.2.2 Bob

Bob receives the initial message from Alice. He verifies the received data, responds with his own message, and finally waits for the last message from Alice to complete the protocol.

3.2.3 Intruder

The Intruder can intercept messages between Alice and Bob. The intruder can store, replay, or modify these messages to try and gain unauthorized information, such as the nonces used by Alice and Bob.

3.3 LTL Verification Formula

The LTL property `mitm_attack` is designed to detect a successful MITM attack. The property is defined as follows:

> *Globally (always), if Alice and Bob both believe they have successfully completed the protocol and each believes the other was their communication partner, then it must be true that the intruder has learned both nonces used in the protocol.*

3.3.1 Formal LTL Specification

The formal LTL specification of the `mitm_attack` property is:

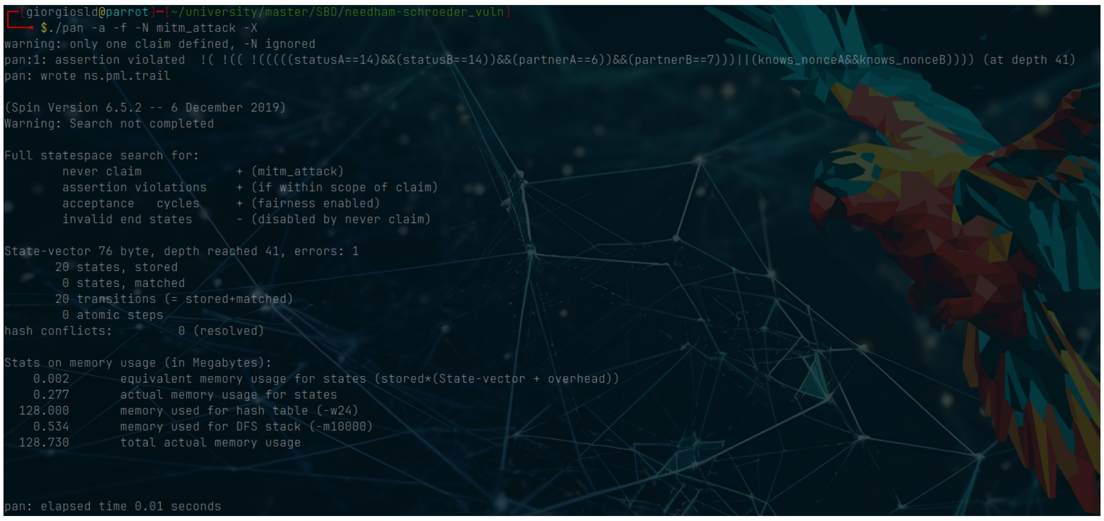
$$\begin{aligned}\text{mitm\_attack} = \Box\Big(&\big(\text{statusA} = \text{ok} \land \text{statusB} = \text{ok} \land \text{partnerA} = \text{agentB} \land \text{partnerB} = \text{agentA}\big) \rightarrow \\ &\big(\neg\text{knows\_nonceA} \land \neg\text{knows\_nonceB}\big)\Big) \qquad (3.1)\end{aligned}$$

Where:

- statusA = ok: Alice believes she has successfully completed the protocol.
- statusB = ok: Bob believes he has successfully completed the protocol.
- partnerA = agentB: Alice believes she was communicating with Bob.
- partnerB = agentA: Bob believes he was communicating with Alice.
- knows_nonceA: The intruder has learned Alice's nonce.
- knows_nonceB: The intruder has learned Bob's nonce.

3.3.2 Verification Process

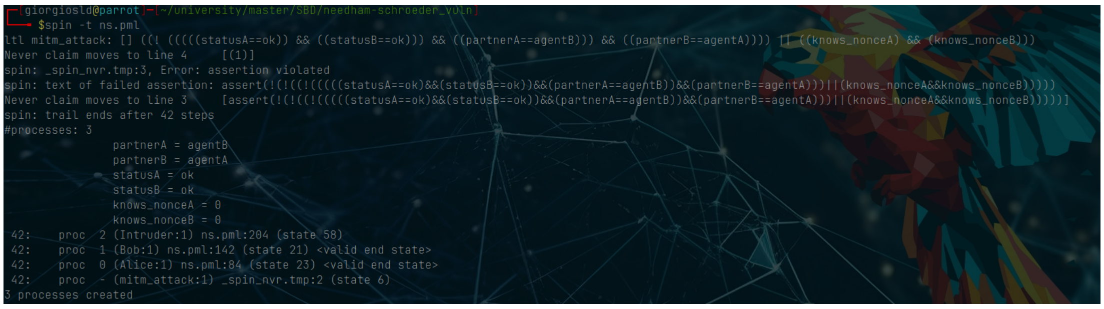
Spin automatically searches for sequences of actions that lead to the violation of the `mitm_attack` property. If such a sequence is found, it indicates a potential vulnerability in the protocol. The verification process identified scenarios where the intruder successfully performed a MITM attack. This confirms the vulnerability in the protocol under the given conditions.



```
┌─[giorgiosld@parrot]─[~/university/master/SBD/needham-schroeder_vuln]
└──╼ $./pan -a -f -N mitm_attack -X
warning: only one claim defined, -N ignored
pan:1: assertion violated  !( !(( !(((((statusA==14)&&(statusB==14))&&(partnerA==6))&&(partnerB==7)))||(knows_nonceA&&knows_nonceB)))) (at depth 41)
pan: wrote ns.pml.trail

(Spin Version 6.5.2 -- 6 December 2019)
Warning: Search not completed

Full statespace search for:
        never claim             + (mitm_attack)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness enabled)
        invalid end states      - (disabled by never claim)

State-vector 76 byte, depth reached 41, errors: 1
       20 states, stored
        0 states, matched
       20 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.002       equivalent memory usage for states (stored*(State-vector + overhead))
    0.277       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage


pan: elapsed time 0.01 seconds
```

Figure 3.1: Execution of ltl property

The verification process identified scenarios where the intruder successfully performed a MITM attack. This confirms the vulnerability in the protocol under the given conditions. The detailed sequence of actions leading to the property violation was analyzed using the trail file generated by Spin.



```
┌─[giorgiosld@parrot]─[~/university/master/SBD/needham-schroeder_vuln]
└──╼ $spin -t ns.pml
ltl mitm_attack: [] ((! (((((statusA==ok)) && ((statusB==ok))) && ((partnerA==agentB))) && ((partnerB==agentA)))) || ((knows_nonceA) && (knows_nonceB)))
Never claim moves to line 4     [(1)]
spin: _spin_nvr.tmp:3, Error: assertion violated
spin: text of failed assertion: assert(!(!((!(((((statusA==ok)&&(statusB==ok))&&(partnerA==agentB))&&(partnerB==agentA)))||(knows_nonceA&&knows_nonceB)))))
Never claim moves to line 3     [assert(!(!((!(((((statusA==ok)&&(statusB==ok))&&(partnerA==agentB))&&(partnerB==agentA)))||(knows_nonceA&&knows_nonceB)))))]
spin: trail ends after 42 steps
#processes: 3
                partnerA = agentB
                partnerB = agentA
                statusA = ok
                statusB = ok
                knows_nonceA = 0
                knows_nonceB = 0
 42:    proc  2 (Intruder:1) ns.pml:204 (state 58)
 42:    proc  1 (Bob:1) ns.pml:142 (state 21) <valid end state>
 42:    proc  0 (Alice:1) ns.pml:84 (state 23) <valid end state>
 42:    proc  - (mitm_attack:1) _spin_nvr.tmp:2 (state 6)
3 processes created
```

Figure 3.2: Trail error from Spin

3.4 Modeling the Fixed Protocol

To address the vulnerability to Man-In-The-Middle (MITM) attacks in the original Needham-Schroeder protocol, we implemented a fixed version of the protocol. The primary modification involves additional checks and the inclusion of cryptographic elements to prevent the intruder from successfully impersonating Alice or Bob.

3.4.1 Modeling the Fixed Protocol in Promela

In the fixed version of the protocol, the following changes were made:

- **Message Authentication**: Added cryptographic checks to ensure message authenticity.
- **Nonce Validation**: Ensured that nonces are validated in each step to prevent reuse by an intruder.

- **Modified definition of Message**: The message definition was modified to fit the new transition between B and A.

Here is a simplified version of the Promela model for the fixed protocol:

```
/* ... (unchanged portions of the Promela model) ... */
/* Encrypted msg.  */
typedef Crypt {
    mtype key;
    mtype content1;
    mtype content2;
    mtype content3;
};

Agent (A)lice
/* Proceed only if the key field of the data matches keyA, the received
nonce is the one that we have sent earlier and if there is auth of agentB */
    if
    :: (data.key == keyA) && (data.content1 == agentB) &&
    (data.content3 == nonceA) ->

Agent (B)ob
    /* Proceed if key field matches keyB and if the agentA sends his
    authentication */
    if
    :: (data.key == keyB) && (data.content2 == agentA) ->
        /* Get Alice's nonce */
        pnonce = data.content1;

        /* Prepare the second message */
        messageBA.key = pkey;
        messageBA.content1 = agentB;
        messageBA.content2 = nonceB;
        messageBA.content3 = pnonce;

/* ... (unchanged portions of the Promela model) ... */
```

3.4.2 Verifying the Fixed Protocol

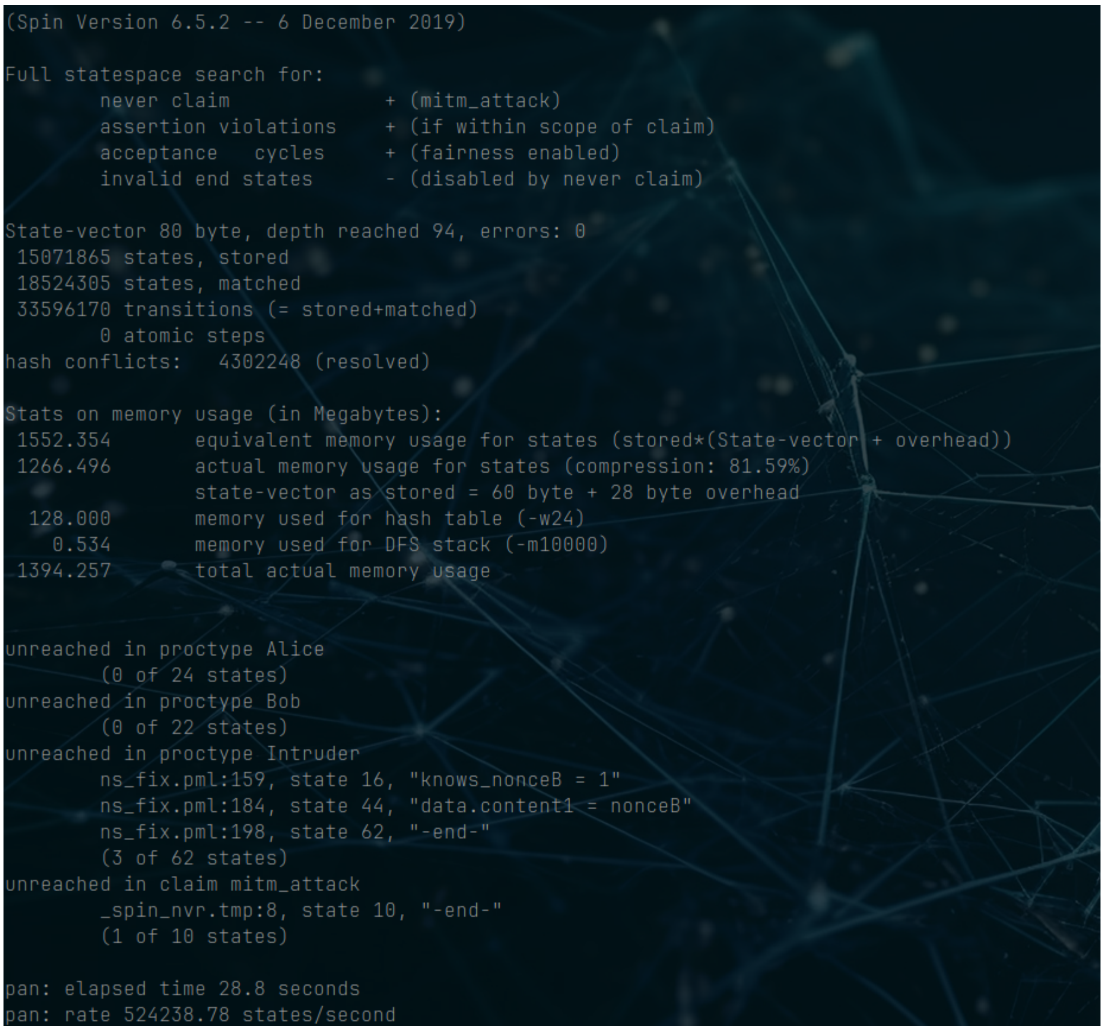
Using the s LTL formula for detecting MITM attacks, we verified the fixed protocol:

$$\begin{aligned}\text{mitm\_attack} = \Box\Big(&\big(\text{statusA} = \text{ok} \wedge \text{statusB} = \text{ok} \wedge \text{partnerA} = \text{agentB} \wedge \text{partnerB} = \text{agentA}\big) \rightarrow \\ &\big(\neg\text{knows\_nonceA} \wedge \neg\text{knows\_nonceB}\big)\Big) \qquad (3.2)\end{aligned}$$

3.4.3 Results

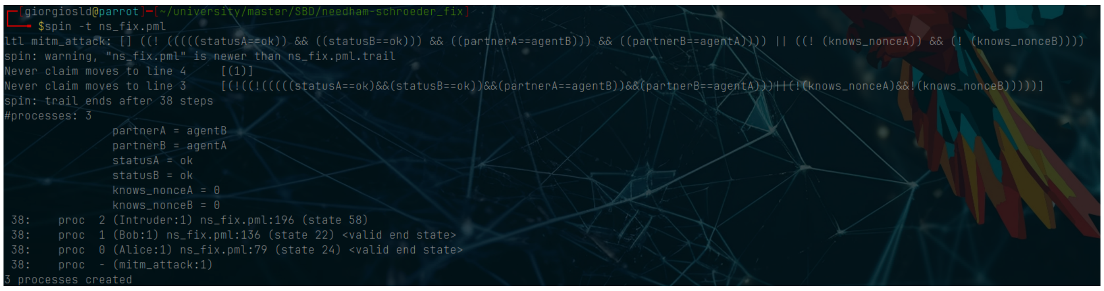
After implementing the enhanced protocol, we employed the Spin model checker to evaluate the new LTL property. The verification results were promising, as the LTL formula was no longer violated, demonstrating that the improved protocol effectively fixed the previously detected MITM attack.

For detailed insights, refer to the screenshots below showcasing the execution and verification process of the enhanced property.



```
(Spin Version 6.5.2 -- 6 December 2019)

Full statespace search for:
        never claim             + (mitm_attack)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness enabled)
        invalid end states      - (disabled by never claim)

State-vector 80 byte, depth reached 94, errors: 0
 15071865 states, stored
 18524305 states, matched
 33596170 transitions (= stored+matched)
        0 atomic steps
hash conflicts:   4302248 (resolved)

Stats on memory usage (in Megabytes):
 1552.354       equivalent memory usage for states (stored*(State-vector + overhead))
 1266.496       actual memory usage for states (compression: 81.59%)
                state-vector as stored = 60 byte + 28 byte overhead
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
 1394.257       total actual memory usage


unreached in proctype Alice
        (0 of 24 states)
unreached in proctype Bob
        (0 of 22 states)
unreached in proctype Intruder
        ns_fix.pml:159, state 16, "knows_nonceB = 1"
        ns_fix.pml:184, state 44, "data.content1 = nonceB"
        ns_fix.pml:198, state 62, "-end-"
        (3 of 62 states)
unreached in claim mitm_attack
        _spin_nvr.tmp:8, state 10, "-end-"
        (1 of 10 states)

pan: elapsed time 28.8 seconds
pan: rate 524238.78 states/second
```

Figure 3.3: Execution of ltl property after fix



```
┌─[giorgiosld@parrot]─[~/university/master/SBD/needham-schroeder_fix]
└──╼ $spin -t ns_fix.pml
ltl mitm_attack: [] ((! (((((statusA==ok)) && ((statusB==ok))) && ((partnerA==agentB))) && ((partnerB==agentA)))) || ((! (knows_nonceA)) && (! (knows_nonceB))))
spin: warning, "ns_fix.pml" is newer than ns_fix.pml.trail
Never claim moves to line 4     [(1)]
Never claim moves to line 3     [(!((!(((((statusA==ok)&&(statusB==ok))&&(partnerA==agentB))&&(partnerB==agentA)))||(!(knows_nonceA)&&!(knows_nonceB)))))]
spin: trail ends after 38 steps
#processes: 3
                partnerA = agentB
                partnerB = agentA
                statusA = ok
                statusB = ok
                knows_nonceA = 0
                knows_nonceB = 0
 38:    proc  2 (Intruder:1) ns_fix.pml:196 (state 58)
 38:    proc  1 (Bob:1) ns_fix.pml:136 (state 22) <valid end state>
 38:    proc  0 (Alice:1) ns_fix.pml:79 (state 24) <valid end state>
 38:    proc  - (mitm_attack:1)
3 processes created
```

Figure 3.4: Spin Trail after Fix

4. UPPAAL

UPPAAL is a robust tool suite for the modeling, simulation, and verification of real-time systems. It uses timed automata to represent system behaviors and properties, allowing for precise modeling of time-dependent aspects. UPPAAL facilitates the formal verification of system models using Computation Tree Logic (CTL), enabling the detection of potential issues such as timing errors and security vulnerabilities.

4.1 Modeling the Needham-Schroeder Protocol in UPPAAL

To model the Needham-Schroeder protocol in UPPAAL, we adapt the Promela model used in SPIN to UPPAAL's syntax and framework. This involves defining templates for the protocol participants (Alice, Bob, and the Intruder), communication channels, and global variables.

4.1.1 Templated and Channels

In UPPAAL, each participant in the protocol is modeled as a template. The communication between participants is handled through broadcast channels, simulating message exchanges.

- **Alice Template:** Represents the initiator of the protocol. Alice sends the first message, waits for a response, and then verifies the received data.
- **Bob Template:** Represents the responder in the protocol. Bob receives the initial message, responds appropriately, and waits for the final message.
- **Intruder Template:** Represents an adversary who can intercept, store, replay, or modify messages between Alice and Bob.

4.1.2 Global Declarations and Variables

The global variables and channels are declared to maintain the protocol state and facilitate communication. These include nonces, keys, and status variables to track the progress of the protocol.

```
// Place global declarations here.
chan start;

// Channels for communication
```

```
chan network;

// Global shared variables for data exchange
int content1;
int content2;

// Nonce and status variables
const int nonceA = 10;
const int nonceB = 11;
const int nonceI = 12;
bool statusA = false;
bool statusB = false;

bool knows_nonceA = false;
bool knows_nonceB = false;

// Key variables and identity
const int keyA = 20;
const int keyB = 21;
const int keyI = 22;
const int agentA = 31;
const int agentB = 32;
const int agentI = 33;

int partnerA;
int partnerB;
int pkey;

// state variable to see if intercept is running in this moment
bool intercept = false;
// state variable to wait that the last message has been sent before start n
bool pround = false;
```

4.2 Verifying the Protocol with CTL

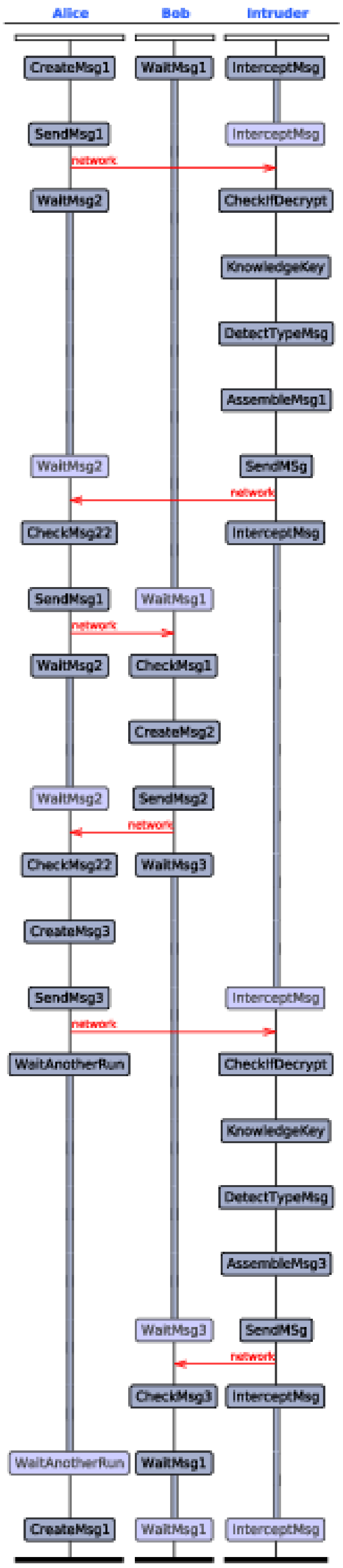
To verify the protocol in UPPAAL, we translate the Linear Temporal Logic (LTL) properties used in SPIN to Computation Tree Logic (CTL) formulas. CTL is used to specify properties over computation trees generated by the model.

4.2.1 Modeling the Vulnerable Protocol

In the vulnerable version, we model the interactions between Alice, Bob, and the Intruder. The intruder intercepts and modifies messages to perform a man-in-the-middle (MITM) attack. The primary CTL property to detect this attack is:

$$\mathsf{A}[\,]((\text{statusA} = 1 \wedge \text{statusB} = 1 \wedge \text{partnerA} = 32 \wedge \text{partnerB} = 31) \implies (\neg\text{knows\_nonceA} \vee \neg\text{knows\_nonceB})) \tag{4.1}$$

This property checks if there exists a path where the intruder learns both nonces.


Alice
Bob
Intruder
CreateMsg1
WaitMsg1
InterceptMsg
SendMsg1
InterceptMsg
network
WaitMsg2
CheckIfDecrypt
KnowledgeKey
DetectTypeMsg
AssembleMsg1
WaitMsg2
SendMSg
network
CheckMsg22
InterceptMsg
SendMsg1
WaitMsg1
network
WaitMsg2
CheckMsg1
CreateMsg2
WaitMsg2
SendMsg2
network
CheckMsg22
WaitMsg3
CreateMsg3
SendMsg3
InterceptMsg
network
WaitAnotherRun
CheckIfDecrypt
KnowledgeKey
DetectTypeMsg
AssembleMsg3
WaitMsg3
SendMSg
network
CheckMsg3
InterceptMsg
WaitAnotherRun
WaitMsg1
CreateMsg1
WaitMsg1
InterceptMsg

4.2.2 Modeling the Corrected Protocol

The corrected version includes additional verification steps to prevent the MITM attack. We ensure mutual authentication by validating the received nonces and the identity of Bob. The same CTL property used before is still valid to verify the corrected protocol.

$$\mathsf{A}[]((\text{statusA} = 1 \wedge \text{statusB} = 1 \wedge \text{partnerA} = 32 \wedge \text{partnerB} = 31) \implies (\neg \text{knows\_nonceA} \vee \neg \text{knows\_nonceB}))$$

This property asserts that for all paths, if Alice and Bob complete the protocol successfully, the intruder does not know either nonce.

4.3 Results and Analysis

After modeling both versions of the Needham-Schroeder protocol in UPPAAL, we ran the model checker to verify the specified properties. The results showed that the vulnerable version allowed the intruder to perform a successful MITM attack, as indicated by the satisfaction of the existential CTL property. For the corrected version, the universal CTL property was satisfied, confirming that the intruder could not learn the nonces when Alice and Bob successfully completed the protocol. These results demonstrate the effectiveness of the protocol modifications in mitigating the MITM attack.

5. Conclusion

This report has presented a comprehensive analysis of the Needham-Schroeder protocol, focusing on its vulnerabilities and the effectiveness of corrections using formal verification tools. The study highlighted the following key points:

- **Vulnerability Identification:** By modeling the original Needham-Schroeder protocol in SPIN, we identified a man-in-the-middle (MITM) attack vulnerability. The SPIN model checker successfully detected this flaw, underscoring the protocol's susceptibility to such attacks.
- **Protocol Correction:** To mitigate the identified vulnerability, a corrected version of the protocol was implemented. This version introduced mutual authentication steps to ensure the integrity of the communication process. The corrected protocol was also modeled in SPIN, and verification showed that the MITM attack was no longer possible.
- **UPPAAL Modeling:** Both the original and corrected versions of the protocol were subsequently modeled in UPPAAL. Adapting the LTL properties used in SPIN to UPPAAL's CTL syntax allowed for the verification of the protocol in a different formalism. This step confirmed the robustness of the corrected protocol in preventing MITM attacks.
- **Verification Results:** The results from both SPIN and UPPAAL consistently demonstrated the presence of the vulnerability in the original protocol and the effectiveness of the corrections. These findings highlight the importance of formal verification in identifying and addressing security vulnerabilities in cryptographic protocols.

In conclusion, the use of formal verification tools such as SPIN and UPPAAL has proven to be invaluable in ensuring the security of communication protocols. This study not only reinforces the necessity of protocol verification but also provides a robust methodology for securing communication protocols against sophisticated attacks. By systematically identifying vulnerabilities and validating corrections, we ensure the reliability and security of protocols like Needham-Schroeder, maintaining their relevance and trustworthiness in modern applications.