Eindhoven University of Technology

Department of Mathematics and Computer Science

Master's Thesis

Classifying Attacks on Security Protocols

by

C. Tankink

Tutor: Dr. F. Dechesne
Supervisor: Dr. S. Andova

Eindhoven, 18th August 2009

Preface

When I started my master in Information Security, my main motivation for this choice was the fact that formal methods, and logic in particular, form a large part in this field: you cannot be sure that a computer system is secure, unless you *prove* it.

While following the courses of the master, my interest in privacy and trust grew, but formal analysis remained my main area of interest, and it would be this are in which I would do my graduation research. So, when I saw an advertisement by Cas Cremers at the ETH Zürich, on "Classification of Attacks on Security Protocols", it caught my interest, and I inquired if it would be possible for me to do my research there. This soon turned out to be impossible, but Cas kindly gave me permission to take the project description and see if I could find someone in Eindhoven who would be interested in supervising me.

As it turned out, there were, and a few weeks later I was having a meeting with three members of the Formal Methods group: Francien Dechesne, Suzana Andova and Simona Orzan. In this meeting, we decided that I would carry out the assignment at the TU/e, and that Francien would supervise me, together with Suzana.

The assignment proved to be as challenging and interesting as I had hoped, not in the least because I was allowed to take it in the direction I found most interesting. For allowing this, and for giving good feeback, I want to thank both Francien and Suzana, who supervised me in an excellent way, asking the right questions and giving the right comments, giving me new ideas on how to continue my research. Also, I would like to thank Jerry den Hartog for being a member on my assessment committee and providing ideas after my midterm presentation.

Next, many thanks go to the members of the Formal Methods group, especially my master student colleagues and my office mates Helle and Paul, for the many interesting discussions over lunch and coffee.

Finally, a big thank you go to my parents and brother, for being a steady support during the project, and trying to understand what I was doing, and keeping interest even if I was rambling too technically.

And last, but most certainly not least, my thanks go out to my girlfriend, Cora, for being a being a moral support during my project, despite being abroad during good portion of it. Cora, I love you, and missed you incredibly during the final stretch. Thank you for being there for me, no matter how far the physical distance.

Carst Tankink
Eindhoven, 18th August 2009

Summary

This thesis investigates a formalized taxonomy for classifying attacks on security protocols. We begin by introducing a trace-based model of attacks, in which traces that are consistent with each other are grouped into runs. These runs form the main elements of an attack, as they are combined by the attacker, who is allowed to modify the messages between two runs.

We formulate our taxonomy based on how runs can coexist structurally, following and extending the taxonomy of Syverson [19]. Next to the structural taxonomy, we also introduce a taxonomy to better capture the actions taken by the attacker in modifying the messages. These actions are represented in a graph-based model, in which messages are connected by edges representing the attacker's action.

To show the applicability of our taxonomy, we use it in the analysis of several attacks against different protocols. To facilitate this analysis, we have implemented a prototype tool automating the classification of attacks.

We conclude that our model can be a useful asset in attack analysis, but should mainly be regarded as the foundations for a larger framework for reasoning about attacks. Furthermore, the prototype can be extended in several ways, including adapting it to suit more than one tool.

Contents

Chapter 1

Introduction

> The openness of the Internet makes it easy for unauthorized persons to approach the gates of computers and computer networks and, if those gates are not properly guarded, to hack through them and gain entry to those computers. Inside, they can read personal and business files out of a morbid curiosity, or, if they are more vicious, to change or even destroy them.
>
> David Kahn, *The Codebreakers* [11]

The quote above sums up one of the major problems of modern-day communication. As networks grow and become more open, the data on these networks get more and more exposed to malicious users. Add to this the fact that the data users are entrusting to the networks (especially the Internet) are becoming more private (including health data and financial information), and it becomes clear that communication between users needs to be secure.

What it means for communication to be secure is open to many interpretations, which translates into *security goals.* There are several security goals, and specific techniques for each of them. The best known are the so called CIA-requirements: Confidentiality, Integrity and Availability.

Confidentiality aims at keeping the exchanged messages secret. This can for example mean keeping personal health data hidden from anyone except the patient and her doctor.

Integrity requires that messages cannot be changed by a malicious third party, or at least that such modifications can be detected. Think, for example, of a requirement that the amount to be transferred cannot be altered before the payment message is received by the bank.

Integrity can also mean that the sender and receiver of a message go unmodified.

Availability means that messages do not get lost in transit, and that a service is available when it is required.

See Anderson's *Security Engineering* [1] for a more complete treatment of security goals and ways of achieving them.

In this thesis, we shall focus on a fourth requirement, which is supplemental to the CIA requirements.

Authenticity is fairly similar to integrity, but also requires that a message cannot be stored and replayed at a later time.

Following Anderson, authenticity is integrity and freshness, meaning that a message is neither forged (integrity) nor replayed from an earlier session (freshness).

To achieve the security goals, a participant can make use of cryptography. Cryptography alone, however, is no guarantee that any of the goals are actually met, since cryptography used in a wrong way could still disclose some vital information. The actual exchange of encrypted messages is called a *security protocol*.

This thesis investigates security protocols on a formal level. On this level, a protocol can be analyzed with the help of tools. This analysis either proves a protocol correct, or provides an indication of why such a proof is impossible to give. Many failed analyses give this indication in the form of an *attack*. An attack is a sequence of messages and actions that show how a malicious user (an *attacker* or *intruder*) can prevent the protocol from achieving its goal.

The attacks returned by a tool, however, do not always give insight in the error occurring in the protocol, including the possibility that the attack is not valid at all. For those attacks that are valid, it is still not always intuitively clear to a protocol designer how the attack relates to a protocol error. To improve this intuitive relation, we try to find a method for reducing the attack to the protocol flaw.

As a step towards this method, we introduce a *classification of attacks*. Such a classification can be either intra-protocol, based on attacks on a single protocol, or inter-protocol, based on attacks that seem similar, but are mounted against different protocols. We opted for the latter, defining a classification describing attacks almost independently of the protocol against which they occur. The protocol does play a small role in modelling the attack, but is not used in any further analysis of the attack. However, we will also see that the defined taxonomy can also be used for looking at intra-protocol classification.

To illustrate several aspects of our theory, we shall often refer to one protocol. This protocol is known as the *Needham-Schroeder Public Key* (NSPK) protocol [16]. The protocol uses public key encryption to obtain authentication. We present the protocol as an informal *protocol narration* here, and give a more formal version later on.

$$\begin{array}{lll}
1. & i \rightarrow r: & \{\!|i, n_i|\!\}_{pk(r)} \\
2. & r \rightarrow i: & \{\!|n_i, n_r|\!\}_{pk(i)} \\
3. & i \rightarrow r: & \{\!|n_r|\!\}_{pk(r)}
\end{array}$$

The protocol is a *mutual authentication* protocol, which is meant to authenticate the responder r to the initiator i and the initiator i to the responder r. Both the fact that the agents participated, and that this participation is recent needs to be proved by executing the protocol. Additionally, the nonces n_i and n_r are supposed to be kept secret during the protocol execution.

The NSPK protocol is not without flaws, as Gavin Lowe pointed out in 1995 (17 years later!) [12]. He showed that an intruder could *attack* the protocol by manipulating the messages of the protocol. We show a narration of the attack here.

$$\begin{array}{lll}
1. & \mathit{Alice} \rightarrow \mathit{Eve}: & \{\!|\mathit{Alice}, N_{\mathit{Alice}}|\!\}_{pk(\mathit{Eve})} \\
1'. & (\mathit{Alice}) \rightarrow \mathit{Bob}: & \{\!|\mathit{Alice}, N_{\mathit{Alice}}|\!\}_{pk(\mathit{Bob})} \\
2'. & \mathit{Bob} \rightarrow (\mathit{Alice}): & \{\!|N_{\mathit{Alice}}, N_{\mathit{Bob}}|\!\}_{pk(\mathit{Alice})} \\
2. & \mathit{Eve} \rightarrow \mathit{Alice}: & \{\!|N_{\mathit{Alice}}, N_{\mathit{Bob}}|\!\}_{pk(\mathit{Alice})} \\
3. & \mathit{Alice} \rightarrow \mathit{Eve}: & \{\!|N_{\mathit{Bob}}|\!\}_{pk(\mathit{Eve})} \\
3'. & (\mathit{Alice}) \rightarrow \mathit{Bob}: & \{\!|N_{\mathit{Bob}}|\!\}_{pk(\mathit{Bob})}
\end{array}$$

The attack starts when *Alice*, playing the role of initiator i, tries to start the protocol with the intruder, *Eve*. The intruder uses this protocol run to make responder *Bob* believe that he is talking to *Alice* (displayed in the protocol narration by (*Alice*)), while the initiator knows nothing about this conversation. This breaks the authentication from *Bob* to *Alice*. Because the intruder places herself[1] between the two honest agents, this attack became known as a *man*[2]-*in-the-middle* (MITM) attack.

Next to an attack, Lowe also gives a suggestion on how to repair the protocol. This is done by replacing the second message with $\{\!|r, n_i, n_r|\!\}_{pk(i)}$. This denies the attacker the reuse of the message in her execution with the initiator (the non-primed messages), since it would show that the origin of message 2 is not *Eve*, but *Bob*.

This example shows that from the attack, it is immediately clear how the protocol can be repaired, or what its actual weakness is, and reinforces our belief that we need some tool in reducing an attack to the flaw in protocol and a recommendation for reparation. Furthermore, we would like the taxonomy to be able to be able to classify an attack into an intuitive class, such as the MITM attack.

1.1 Assignment

The question which this thesis investigates is: "How can we express the similarities in attacks on different protocols, and how can we find a reparation based on these similarities?"

Since there are many different types of protocols, all with their own specific goals, we have decided to narrow the scope of our investigation by only considering attacks against protocols for *authentication*.

We divide the main question into the following subproblems:

- What similarities are there between different protocols?
- How can we capture these similarities in a formal model?
- Can we define a reparation to a protocol based on the classification of an attack against this protocol?

[1]Traditionally, cryptological research personifies the agent names. This led to the intruder (or eavesdropper) being named Eve, and we therefore refer to the intruder as a she.

[2]This already contradicts the fact that the intruder is female, but we shall stick to convention here.

1.2 Approach

This thesis is not a chronological report of the research that lies at its basis. We will present a short chronological report here.

The initial research consisted of a combination of a literature study, bringing up the article of Syverson [19], and an investigation of the SPORE library [18], and especially its formalizations in Scyther [5]. Initially, our research focused on the MITM attack, as this is an intuitively clear attack, and we could find the essential properties of this attack. Especially, we noticed that each attack had both structural properties and properties describing the modifications an attacker made to the messages exchanged between agents. We therefore decided that our taxonomy should consist of two subtaxonomies: one for structural classification, and one for classification of modifications.

We started with formalizing the taxonomy found in Syverson's article, which described a structural taxonomy of attacks. This taxonomy can be found in Chapter 3. The main challenge in formalizing this part of the taxonomy was finding a model for protocols which fitted naturally with the actions of the attacker and the beliefs of the honest agents. As we shall show in Chapter 2, this is captured in the notion of protocol *runs*.

Having formalized the structural classification, we implemented a prototype tool that could classify the attacks based on this taxonomy. This implementation showed that it is possible to build a tool automatically classifying attacks. When run on the SPORE formalizations, it also showed that a lot of attacks were structurally the same (interleaved attacks), while the modifications in between were different. This enforced our belief that a taxonomy for modifications was necessary.

To arrive at the taxonomy of modifications described in Chapter 4, we first tried to model them as functions. This already proved to work for MITM attacks, but was flawed for more complex attacks, in which multiple messages were combined or in which the order of messages was changed. Furthermore, the taxonomy could not capture the actions of the intruder, but only showed the results of those actions. We decided to capture these properties by building graphs, and building a classification for these graphs.

During and after forming the model, we have studied several attacks. We decided to gather the results of these studies in a single chapter (Chapter 5). These results range from our initial results on the MITM attack to using the model to reason about equality of attacks and reducing attacks to other, clearer attacks. We also use the model to prescribe some generic reparations based on attack classification.

1.3 Reader's guide

This thesis starts with introducing a model for protocols and attacks in Chapter 2. As we shall show in that chapter, the taxonomy along which we classify attacks is composed of two different subtaxonomies, a structural taxonomy and a taxonomy of modifications. The structural taxonomy is defined in Chapter 3, the modification taxonomy in Chapter 4. To show how the framework of taxonomies can be used, we apply our theory to some attacks found on protocols in the Security Protocols Open Repository (SPORE) [18]. This application is shown in Chapter 5. The experiments on SPORE were carried out with a prototype implementation in Python. This implementation is described in Chapter 6, while the program code can be found in Appendix A.

We draw our conclusions in Chapter 7.

Chapter 2

Protocol and Attack Model

In this chapter, we define a model for security protocols and for attacks on these protocols.

2.1 Protocols

A security protocol is an exchange of messages between two or more agents. The messages can be enriched with cryptographic operations, like encryption and digital signatures.

In this thesis, we focus on the analysis of such protocols on a high and formal level, studying the exchange of messages between agents, and disregard attacks exploiting flaws in the cryptographic algorithms or in the implementation. That is, we assume a *black box* model of cryptography [8].

2.1.1 Terminology

To have a common base of knowledge, we shall introduce the following terms informally. Of these, the term protocol is the only one that will be defined more formally later on.

Agents Agents are the executors of protocols. These agents are actually (programs running on) computer systems, but we shall "personify" the agents by giving them the names traditionally used in security literature.

There are two different kinds of agents. The honest agents execute a protocol as intended, while a dishonest agent is acting on behalf of the attacker.

Honest agents are named *Alice*, *Bob*, *Charlie* and *Dave*, while a dishonest agent is named *Eve*. A server used as a trusted third party in communication goes by the name of *Simon*.

Role A role is a description of the actions taken by one agent in a protocol. Several roles make up one protocol description. A role has a name describing the part of the protocol it plays, for example: Initiator, Responder or Server. We shall often abbreviate these roles with the letters i, r and s.

Protocols As already mentioned, protocols are exchanges of messages between agents. In the literature, there are two occurrences of the term "protocol", between which we need to distinguish.

The first are *protocol descriptions.* These descriptions describe how a protocol should play out, in an ideal setting, where all agents are honest. This description is constructed from one or more (typically two or three) roles.

The second are *instantiations of protocols.* These instantiations represent possible exchanges of messages for a given protocol description. We shall call a correct installation of a protocol, in which the messages exchanged are according to the description, a protocol *run.*

Multiple runs can be combined to create an *attack.* An attack is an execution of a protocol in which, from the viewpoint of the attacked agent, the goal of the protocol is fulfilled, while this was not actually the case, due to intermission of the intruder.

We will formally define an attack later in this chapter.

2.1.2 Protocol model

Our analysis is based on a model for protocols which represents protocol executions as traces of agent actions. This model, known as the traces model, is similar to the model used by Cremers [6].

In such a model, analysis is based on exploring the potential instantiations of protocol traces and attacks are given as traces violating the goals of the protocol.

We use the following sets of basic primitives:

- *Nonces (Nonce)* are random values that are generated by an agent. We assume that the randomness is perfect, implying that it is impossible to guess the value of a nonce before it is generated, and impossible to get the same nonce twice.

- *Variables (Var)* are used in a protocol description to signify values obtained from the network. In particular, the protocol roles are variables, instantiated with agent names during a run of the protocol. These role variables comprise their own subset of *Var*, namely *Role*

  In an actual run, each variable is instantiated to some value. We assume that a run by an honest agent uses the same instantiation for the variable. For example, assume two messages $m_1 = \{\!|n_i|\!\}_{sk(i)}$ and $m_2 = \{\!|n_r|\!\}_{pk(i)}$ are part of a protocol description, for variables n_i, n_r and i. For a honest agent, if i is instantiated as *Alice* in m_1, it should also be instantiated to *Alice* in m_2. n_i and n_r do not have this restriction, as they are different variables.

- *Constants (Const)* are global constants. These values include agent names and keys, and represent everything that is not generated during a protocol run. A protocol run only contains constants and nonces.

  In this thesis, we shall assume that keys are represented by functions, which can be applied to zero (for sessions keys generated by an agent), one (for public and private keys) or two (for shared, symmetric keys) agent names.

We build our protocol model incrementally with these elements, determining which requirements should hold for which parts of the protocol.

Definition 2.1 (Term)**.** A term is constructed from basic primitives, by tupling, encrypting or applying.

This means we have the following construction rules for terms

$$\mathit{BasicTerm} ::= \mathit{Nonce} \mid \mathit{Var} \mid \mathit{Const}$$

$$\mathit{Term} ::= \mathit{BasicTerm} \mid (\mathit{Term}, \mathit{Term}) \mid \{\!|\mathit{Term}|\!\}_{\mathit{Term}} \mid \mathit{Term}(\mathit{Term})$$

The constructor for encryption, $\{\!|m|\!\}_k$ (for some terms m and k), covers both symmetric and asymmetric encryption. To this end, we require the key k to have an inverse k^{-1}. For symmetric keys $k^{-1} = k$. Since we assume perfect cryptography, it is only possible to decrypt a term $\{\!|m|\!\}_k$ if one is in the possession of k^{-1}.

We shall adopt the following conventions for encryption and decryption:

- $pk(A)$ is the public key of agent A. This key is known to all agents, including the intruder.
- $sk(A)$ is the secret key of agent A. This key is only known to A, unless it leaks during the execution of the protocol. For decryption, we have $pk(A)^{-1} = sk(A)$ and $sk(A)^{-1} = pk(A)$, so terms encrypted under A's public key can only be decrypted using the secret key of A, and messages encrypted with the secret key of A can only be decrypted using the public key of A.
- $k(A, B)$ is a symmetric key shared between agents A and B. This key is known only to A and to B, unless leaked. We have that $k(A, B)^{-1} = k(A, B)$.

When a protocol is analyzed, the attacker is initially assumed to be in the possession of a set of knowledge $\mathcal{K}$. This set consists of all data that is public, such as the agent names and any public keys. Following our convention, this means that the attacker knows the function pk, and can apply it to any agent name. Furthermore, the attacker knows the result of the sk function for the dishonest agent, *Eve*. In the case of secret key encryption, the intruder knows any key shared with *Eve*.

These terms are the principal elements of a protocol, and sending and reading them are the principal actions of an agent. This is captured in the notion of *events*. An event represents a message being sent from a *source* to a *target*.

Definition 2.2 (Event)**.** An event is the action of sending or receiving a term.

$\mathit{send}(R_1 \to R_2 : T)$ Signifies that role R_1 should send a term T, intended for R_2.

$\mathit{read}(R_1 \to R_2 : T)$ Signifies that role R_2 receives a term T, apparently from R_1.

In both cases, the arrow and the colon are syntactic sugar, to suggest the style of protocol narrations.

Instead of roles, an event can also contain agent names as source and target.

We combine the events into a trace, by imposing an order on the events, together with a notion of the role executing the trace.

Definition 2.3 (Trace)**.** A trace is a 3-tuple $(R, \mathcal{E}, \leq)$, where:

R of type *Role* is the name of the role executing the trace.

$\mathcal{E}$ is set of events, as described in Definition 2.2. These events are the events executed in the trace

$\leq$ is a partial order on $\mathcal{E}$.

As a trace depicts the "viewpoint" of a role R in the protocol description, we require that the events of a trace are consistent with the R it is written for. In other words, we require the following predicate to hold for any trace T:

$$(\forall e \in T.\mathcal{E} \cdot e \equiv \mathit{send}(R_1 \to R_2 : m) \Rightarrow R_1 = T.R \\ \wedge\, e \equiv \mathit{read}(R_1 \to R_2 : m) \Rightarrow R_2 = T.R)$$

A protocol is defined by combining a set of traces, one for each role intended to be played in the protocol. The role name of each trace should be unique in this collection. Furthermore, a relation C between trace events is also given. This relation (the communication) determines which *send* event corresponds to which *read* event.

Definition 2.4 (Protocol)**.** A protocol is a tuple (Ts, C), with the following elements:

Ts is a set of traces.

C is a binary, symmetric relation on events occurring in different traces in Ts.

The following constraints should hold for any protocol, assuming we can obtain the elements of a tuple by their names:

- Each role occurs at most once: $(\forall t_1, t_2 \in Ts : t_1.R = t_2.R \Rightarrow t_1 = t_2)$.
- The target, source and message of each event matched by C should be equal:

$$(\forall (e_1, e_2) \in C : e_1 \equiv \mathit{send}(R_1 \to R_2 : m) \Rightarrow e_2 \equiv \mathit{read}(R_1 \to R_2 : m) \wedge \\ e_2 \equiv \mathit{send}(R_1 \to R_2 : m) \Rightarrow e_1 \equiv \mathit{read}(R_1 \to R_2 : m))$$

- The links established by C should preserve the relation $\leq$ of the individual threads:

$$(\forall T_1, T_2 \in Ts.(\forall e_1, e_1' \in T_1, e_2, e_2' \in T_2. \\ (e_1, e_2) \in C \wedge (e_1', e_2') \in C \Rightarrow e_1 \leq e_1' \Leftrightarrow e_2 \leq e_2'))$$

Example 2.5 (Needham-Schroeder Public Key protocol)**.** As an example of the definitions above, we shall formulate the NSPK protocol [16] in the model defined thus far.

Terms The terms used in NSPK are:

- $\{\!|i, n_i|\!\}_{pk(r)}$
- $\{\!|n_i, n_r|\!\}_{pk(i)}$
- $\{\!|n_r|\!\}_{pk(r)}$

Here, i and r are role names, pk is a constant function, and n_i and n_r are both nonces and variables, depending on the event in which they occur.

Events The following events use the terms defined above.

- $send(i \to r : \{\!|i, n_i|\!\}_{pk(r)})$
- $send(r \to i : \{\!|n_i, n_r|\!\}_{pk(i)})$
- $send(i \to r : \{\!|n_r|\!\}_{pk(r)})$
- $read(i \to r : \{\!|i, n_i|\!\}_{pk(r)})$
- $read(r \to i : \{\!|n_i, n_r|\!\}_{pk(i)})$
- $read(i \to r : \{\!|n_r|\!\}_{pk(r)})$

When n_i (or n_r) is part of a read event, it is a variable. When it is part of a send event, it is a nonce.

Traces There are two traces defined for this protocol. One for the initiator (role i) and one for the responder (role r).

1. R**:** i

   $\mathcal{E}$**:** $\{send(i \to r : \{\!|i, n_i|\!\}_{pk(r)}),$
   $send(i \to r : \{\!|n_r|\!\}_{pk(r)}), read(r \to i : \{\!|n_i, n_r|\!\}_{pk(i)})\}$

   $\leq$**:**
   - $send(i \to r : \{\!|i, n_i|\!\}_{pk(r)}) \leq read(r \to i : \{\!|n_i, n_r|\!\}_{pk(i)})$
   - $read(r \to i : \{\!|n_i, n_r|\!\}_{pk(i)}) \leq send(i \to r : \{\!|Nr|\!\}_{pk(r)})$

2. R**:** r

   $\mathcal{E}$**:** $\{read(i \to r : \{\!|i, n_i|\!\}_{pk(r)}),$
   $read(i \to r : \{\!|n_r|\!\}_{pk(r)}), send(r \to i : \{\!|n_i, n_r|\!\}_{pk(i)})\}$

   $\leq$**:**
   - $read(i \to r : \{\!|i, n_i|\!\}_{pk(r)}) \leq send(r \to i : \{\!|n_i, n_r|\!\}_{pk(i)})$
   - $send(r \to i : \{\!|n_i, n_r|\!\}_{pk(i)}) \leq read(i \to r : \{\!|n_r|\!\}_{pk(r)})$

Protocol The complete protocol consists of the above two traces, and a relation C. C pairs the *read* and *send* events of equal terms t to each other. We will not enumerate these pairs.

The protocol is depicted in the message sequence chart (MSC) in Figure 2.1. In this figure, the *send* and *read* events paired by C are depicted as an arrow from the source to the target.

In the rest of this thesis, we shall not mention the actual protocol definitions, when this is not necessary. Instead, we shall use the visualization in an MSC.

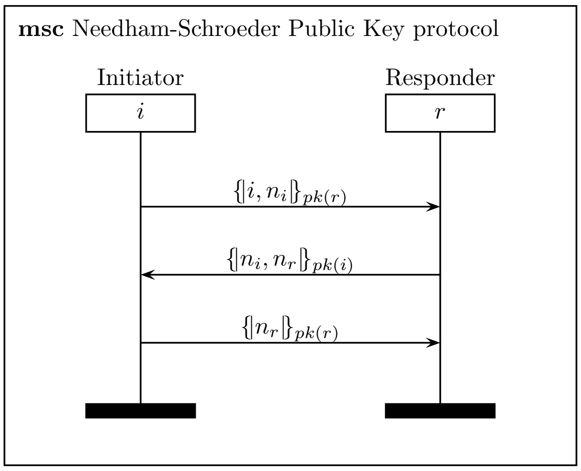


Figure 2.1: The Needham-Schroeder Public Key protocol

2.1.3 Run model

The instantiation of a protocol represents an actual "execution" of that protocol. In these instantiations, honest agents take up the protocol roles, while the intruder, possibly using a dishonest agent as a representative, modifies the messages in transit.

Instantiation of an event occurs by substituting the role names of the event with an agent name, and instantiating each variable within the message of the event with a constant or a nonce. We will only define this instantiation as part of the instantiation of traces in the form of threads.

Definition 2.6 (Thread)**.** A thread is a tuple (T, ν), where:

T is a trace.

ν is a function mapping the term variables to constants: $\nu : Var(T) \to Const \cup Nonce$. $Var(T)$ is a function returning the variables of all events in T. This includes the role names occurring as the sources and targets of events.

In the same way as traces are combined into protocols, threads are combined into runs. Actually, a run represents the execution of a protocol that is consistent between the agents executing it, and the attacker is allowed to modify the events between the elements.

Formally, a run is an instantiation of a protocol, consisting of a single thread for each trace of the protocol.

Definition 2.7 (Run)**.** A run is a tuple (P, Ths), with:

P is the protocol being run.

Ths is a set of of threads.

The following constraints should hold:

- Threads are instantiated from traces of the protocol:

$$(\forall th \in Ths : (\exists t \in P.Ts : th.T = t))$$

- Threads are instantiated at most once from a trace:

$$(\forall th_1, th_2 \in Ths : (\exists t \in P.Ts : t = th_1.T \wedge t = th_2.T) \Rightarrow th_1 = th_2)$$

- The instantiations of the threads are consistent among each other, so all variables instantiated are instantiated equally:

$$(\forall th_1, th_2 \in Ths \cdot (\forall v \in Var(th_1.T) \cap Var(th_2.T) \cdot th_1.\nu(v) = th_2.\nu(v)))$$

For simplicity in the rest of this thesis, we let $e \in r$, for some event e and run r, denote that event e occurs in the threads of r. For the sake of legibility, we opted not to define this in a more formal way.

2.2 Verification of protocols

The only way to be sure that a protocol works as intended, is by rigorous verification. This verification is normally done formally, in a model representing the effect each message has on the knowledge of the agents.

In the end, the complete exchange of messages should establish the intended security goal of the protocol, despite the presence of an intruder. In the verification of a protocol, we intend to prove that this claim holds, for a given protocol, goal and intruder.

For the analysis, we adopt the idealized model of cryptography based on the principles set out by Dolev and Yao [8]. This model consists of the following general assumptions:

- Cryptography is perfect.
- The attacker is the network.

Cryptography is perfect

Perfect cryptography means two things:

- It is impossible to obtain any information about plaintext messages or keys from the ciphertext message.
- Without the proper key, it is impossible to obtain the plaintext message from a ciphertext message.

The attacker is the network

We assume that the intruder has full control over the network used, meaning she can intercept and read each message sent, and deliver any message known to her to any agent in the network at any moment.

We can translate these assumptions into the following set of rules about which terms an attacker can read during the execution of a protocol:

- Any message sent by an honest agent is read by the attacker.

- Any message in the attacker's initial knowledge can be read by the attacker.
- Any constant term, with the exception of decryption keys of honest agents, is in the initial knowledge.
- Any decryption key of a dishonest agent is in the initial knowledge.
- Any nonce generated by the attacker is in the initial knowledge.
- Any message read by the attacker can be withheld from the intended recipient.
- Any message read by the attacker can be forwarded to any recipient.
- Only when the attacker has read an encrypted message $\{|m|\}_k$ and the corresponding decryption key k^{-1}, she can read the message m.
- Any message read by the attacker can be encrypted with any message read by the attacker, leading to an encrypted message read by the attacker.

2.2.1 Authentication goals

As already stated, the intruder follows the Dolev-Yao rules and the goal that we focus on here is *authentication.*

In fact, there is not one single notion of authentication, but there are several refinements and corresponding formalizations of the concept. All these notions, however, involve the belief of one agent about the state of another agent, based on the messages exchanged thus far. This belief is defined more precisely by Diffie, Van Oorschot and Wiener [7] in their concept of matching runs. Their definition includes the agents recording a run, which is similar to our definition of threads (Definition 2.6).

As the definition of matching threads requires the matching of messages, we will first describe this notion in our model.

Definition 2.8 (Matching messages)**.** For two threads Th and Th' of the same protocol, messages $m \in Th$ and $m' \in Th'$ *match* if the following conditions hold:

- If m is sent in Th, then m' is received in Th'.
- If m' is sent in Th', then m is received in Th.
- m and m' are equal.

Now that we know when two messages match, we can extend this to the messages exchanged during a protocol run.

Definition 2.9 (Matching threads)**.** Two threads match if:

- The agents executing the respective threads agree on each other's identity.
- Their messages can be partitioned in sets of matching messages, each set containing one message from each thread.

- The messages sent by one agent appear in the same order in both threads.
- The messages sent by the other agent appear in the same order in both threads.

Intuitively, two threads match if the order of the messages has not been modified in transit. So, if messages m_1 and m_2 have been sent by one agent in a particular order, they arrive at the other agent in the same order.

This definition does not require the messages of two agents to be in the same order with respect to each other, so a thread representing events ordered $send(A_1 \rightarrow A_1 : m_1) \leq read(A_2 \rightarrow A_1 : m_2)$ matches with a thread representing the same events, but in the order $send(A_2 \rightarrow A_1 : m_2) \leq read(A_1 \rightarrow A_2 : m_1)$.

In the definition of Diffie, Van Oorschot and Wiener, this definition is split over two "properties of a successful run". We merge it here in one definition, to allow for easier comparison with different authentication properties.

To illustrate the variety of authentication types, we summarize the "Hierarchy of Authentication Specifications", described by Lowe [14]:

Aliveness

When an agent, say Alice, accepts that another participant, say Bob, is alive, she concludes from the messages received and sent thus far, that Bob has once been participating in the protocol.

We can adapt Definition 2.9 to fit aliveness by dropping the requirement that the two threads belong to the same run. It is sufficient that there exists a matching thread by the other agent. Furthermore, it is not necessary that the messages are delivered in order.

Weak agreement

Weak agreement is achieved when two agents complete a run of the protocol, and agree on each other's identity. This does not require the agents to agree on the roles played. This form of authentication is equivalent to that of Definition 2.9.

Lowe, however, claims that weak agreement is stronger than the notion of matching threads, arguing that Definition 2.9 does not force the responding agent to accept the initiating agent's identity. However, as we already noted, two threads only match if the two agents executing the threads *accept each other's identity*, a property which was introduced separately in the Diffie, Van Oorschot and Wiener paper. So the two properties are equally strong.

Non-injective agreement

A protocol guarantees non-injective agreement, if both parties have completed a run, agree on each other's identity and on the role they have played in the protocol. In Lowe's definition, the parties should also have a matching set of data. In practise, this would mean that the key(s) accepted by both parties should be equal.

This is a bit stronger than the definition of matching threads, since it also requires the roles to match.

(Injective) agreement

Agreement is similar to weak agreement, with the exception that for each thread by one agent, there should be one thread by the other agent. In our model, this corresponds to the threads of all agents being in one run. For weak agreement, this is not necessary, and there could be more threads than those of one run.

2.2.2 Verification tools

There are several tools and techniques for verifying a protocol. We shall look at those that use some form of the model described above, and shortly describe other techniques used in protocol verification.

Scyther

The Scyther tool [5] has been and continues to be developed by Cremers.

The tool uses the traces model, calculating a so-called "complete characterization" of a security protocol [6], and matching this characterization to a description of traces violating the property. If there are matching traces in the characterization, these traces are given as an attack.

Next to storing offending traces as XML files, the tool also allows the algorithmic backend, which is supplied as a precompiled binary, to be called from any Python script, supplying this script with objects representing the protocol and the attacks. This allows for easy manipulation of the attack traces. It is therefore a good candidate for experimentation and as a basis of a larger framework.

Another pro is the fact that most of the protocols in the SPORE library [18] are formalized in Scyther's syntax, giving a large test base.

Scyther cannot currently check for injectivity, which means that replay attacks cannot be analyzed with this tool.

ProVerif

ProVerif [2] is a tool which transforms a protocol description and the property to be checked into a set of Horn clauses. Using these clauses, the system tries to prove that an attacker can achieve an unwanted effect, for example, knowing a term noted as secret. If this is the case, the system can construct a trace from the branches of the proof tree. Such effects are specified together with the protocol description.

ProVerif does allow checking for injectivity, and produces replay attacks. The downside is that the tool only outputs the actions by honest agents, and does not display intruder actions. Furthermore, it is not as easy to manipulate the traces as in Scyther: the tool only outputs to standard output. However, as the tool is open source, it should be possible to adapt or use the code to facilitate manipulating attack traces.

Athena

The Athena tool [17] is a tool based on the Strand Space theory [21]. This theory has many similarities to the traces model, but as Athena is not publicly available, we do not consider it further.

Other verification methods

It is also possible to verify a protocol without exploring the entire state space, but by giving a proof that each message of the protocol provides knowledge to the receiving agent about the state the other agent is in. Such proof-based approaches were pioneered by Burrows, Abadi and Needham in the "BAN-logic" named after them. [4].

In their approach, each message is "idealized" into a proposition describing what knowledge can be obtained from it, according to a set of inference rules.

From the formal definition of the protocol one can then, by hand, derive that an agent believes a certain fact, such as: "Message m is fresh, and was sent by agent B", leading to the conclusion that B was *alive*.

There are several problems with the BAN-logic, but the problem making it unfit for our purposes is the fact that when a protocol is not proved correct, then no attack trace is given. The same issue holds for other proof-based systems.

Another (automated) proof-based approach is adopted by the Cryptyc tool [10]. By this so-called type-based approach, a protocol designer is forced to annotate protocol elements with their intended type. Then, protocol is safe, if it can be proved well-typed.

There are several variants of the type-based protocols, including one especially meant for authentication designed by Bugliesi, Focardi and Maffei [3]. These techniques do not generate an attack against a protocol, and are therefore not used to implement our framework. Some of the abstractions these attacks make, however, can be considered when abstracting from attacks.

We decided to use the Scyther tool as a basis for our analysis, as it is freely available, uses the traces model, and can be used in a larger framework. We have tried to adapt the ProVerif tool to analyze replay attacks, but since this is a non-trivial task requiring quite some knowledge about the tool's implementation, we refrained from it, first focusing on implementing the model to work with Scyther.

2.3 Attacks

When a protocol does not fulfil its goal(s), we can, in the trace-based approach, construct a counterexample witnessing this. Such a counterexample shows an actual instantiation of the protocol, in which the attacker takes some steps, and in which one agent believes the goal is met, while the protocol instantiation contradicts this.

For example, when a protocol should satisfy aliveness for role I, the agent playing this role, say *Alice*, believes that she has talked to another agent, say *Bob*. A counter example to this protocol could be an instantiation in which *Alice* runs the protocol, believing that *Bob* was sending the messages she received, while in fact, the intruder constructed the messages, not involving *Bob* in the construction at all. We call these instantiations violating the protocol goals *attacks*.

Attacks are always constructed with respect to one agent in a run, and constructed to break a specific goal. Note that an attack against a weaker protocol goal, is also sufficient to break a stronger one.

We are not really interested in the goals themselves, as the attacks are results of analyzing the protocol goals. We will try to link the attack classification to specific protocol goals, but the goals themselves will not be part of this classification.

2.3.1 Attack model

An attack is a combination of runs. Each of these runs is, from the viewpoint of the agents executing it, a consistent instantiation of the protocol being executed. However, the attacker is allowed to modify messages *between* runs. This modification is captured by a set of graphs, which describe how each message received by an honest agent in a run has been constructed.

Not any combination of runs is an attack, as it is possible for an attack to act just as a router, forwarding messages without being able to modify them. Instead, an attack is a combination of runs, one which is the run being attacked. Based on this run, the (honest) agent executing it can construct a matching thread with another agent, such that the two threads together follow Definition 2.9. The thread constructed this way, is not actually part of the same run. Instead, the messages used in construction of the thread were provided by the attacker, based on some messages occurring in another run.

This construction could have been made by the attacker combining her knowledge to obtain a message, or something as simple as forwarding a message.

Definition 2.10 (Attack)**.** An attack is a tuple $(Rs, \mathcal{G})$, where:

Rs The set of runs used for the attack.

$\mathcal{G}$ The set of modification graphs used for constructing the messages received by the agents.

Example 2.11 (Lowe's attack on NSPK)**.** As an example, consider Lowe's attack on NSPK. This attack is depicted in Figure 2.2.

In the attack, we can recognize two runs. Run one is executed by *Alice* playing the role of initiator i and *Eve* playing the role of responder r. The threads this run are defined by the traces for each role, as found in Example 2.5. The instantiation ν of both threads is:

- $\nu(i) = Alice$
- $\nu(r) = Eve$
- $\nu(n_i) = N_{Alice}$
- $\nu(n_r) = N_{Bob}$

Note that there is no instantiation for pk, as this is a constant of the protocol

Run two consists only of a thread by *Bob*, which is instantiated from the responder trace of NSPK. This thread has the following instantiation ν.

- $\nu(i) = Alice$
- $\nu(r) = Bob$

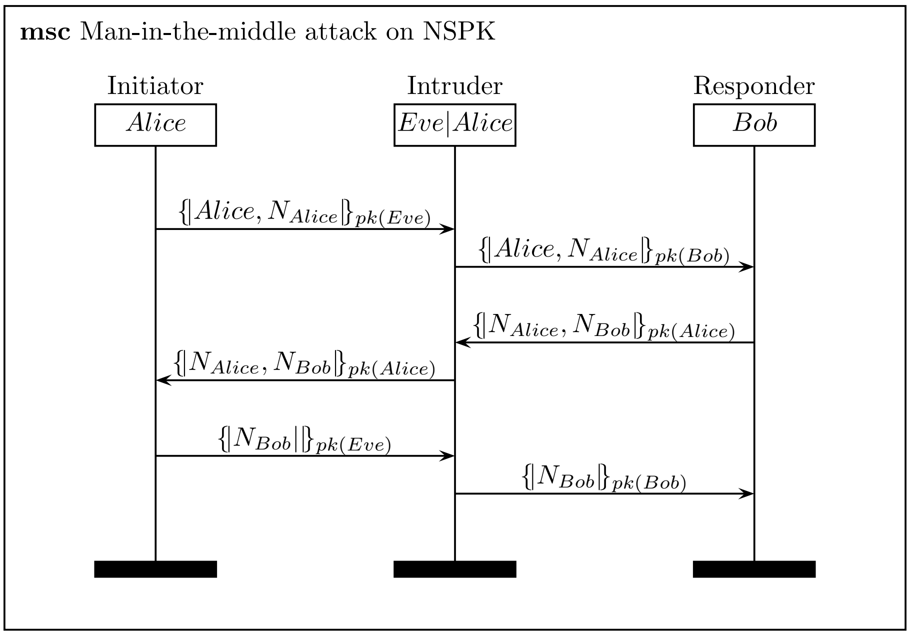

msc Man-in-the-middle attack on NSPK
Initiator
Alice
Intruder
Eve|Alice
Responder
Bob
$\{|Alice, N_{Alice}|\}_{pk(Eve)}$
$\{|Alice, N_{Alice}|\}_{pk(Bob)}$
$\{|N_{Alice}, N_{Bob}|\}_{pk(Alice)}$
$\{|N_{Alice}, N_{Bob}|\}_{pk(Alice)}$
$\{|N_{Bob}||\}_{pk(Eve)}$
$\{|N_{Bob}|\}_{pk(Bob)}$

Figure 2.2: Lowe's attack on NSPK

- $\nu(n_i) = N_{Alice}$
- $\nu(n_r) = N_{Bob}$

The two instantiations disagree on the instantiation of r. As this is one of the protocol roles, we know that this is an attack against authentication. Furthermore, *Bob* executes the attacked run, since this run does not instantiate all traces of the protocol.

What is more, we can immediately see that this attack does not break aliveness: *Bob* thinks he executes a run with *Alice* (shown by $\nu(i) = Alice$), and *Alice* actually has a run of the protocol. The next weakest level of authentication is weak agreement, and it is this property that is broken by the attack: *Alice* does not know that *Bob* participated in the protocol, and can therefore accept his identity.

2.4 Classifying attacks

We now have the equipment necessary to classify attacks on security protocols, by building a taxonomy for possible properties of an attack.

We can build a taxonomy for attacks by looking at the two different elements of the attacks, the runs $A.Rs$ and the modifications $A.\mathcal{G}$.

These elements are independent of each other: one modification g can be used for different combinations of runs, while a single run combination can have different modifications between them.

This leads us to believe that if we can construct a taxonomy for the runs and one for the modifications of an attack, we can use both taxonomies in classifying attacks.

We will formulate the two taxonomies in the next two chapters, and use them for classifying and analyzing some attacks in Chapter 5.

Chapter 3

Structural Classification of Attacks

This chapter describes a structural taxonomy for classifying attacks. The taxonomy describes how the elements of the set Rs of an attack can interact. It does not describe the possible modifications occurring in the attack (the structure of the graph $\mathcal{G}$). That will be the subject of the next chapter.

3.1 A structural taxonomy

For our structural taxonomy, we follow "A Taxonomy of Replay Attacks", by Syverson [19]. In this paper, Syverson describes how one can categorize attacks by differentiating on the modification of message source and target. Combining these two taxonomies, he obtains a final taxonomy.

We make two adaptions to the taxonomy, and formalize it in our model.

Firstly, the original taxonomy refers to replay attacks on the level of subterms. This means that it describes how a part of a message is replayed, to be received by one of the honest agents. We lift the classes described by Syverson to the level of runs, describing how the runs of an attack interact. This decision was made because the main goal of our taxonomy is to classify attacks so that they can be grasped more intuitively. When one tries to classify the origin of every term read, clarity might be lost. Even with this generalization, we find that it is possible to formalize the taxonomy obtained as such in our model.

Secondly, we add a new level of distinction to the taxonomy: a taxonomy describing the difference between message sender between runs.

We will shortly describe the categories of the complete taxonomy here. For a more extensive description, as well as some examples, we refer to Syverson [19].

3.1.1 Taxonomy of origin

The first type of distinction can be made by looking at where the messages of one run are delivered, with respect to their origin. This part of the taxonomy is therefore called the taxonomy of origin.

At the highest level of this taxonomy, there is a distinction between *internal* and *external* attacks.

Internal attacks are attacks in which all messages are taken from inside the current run. This means that the attacker breaks authentication, without needing an extra run.

Recalling the definition of a run, we see that this requires the participating agents to agree on the instantiation of messages and roles. This means that internal attacks are only capable of breaking aliveness, and not one of the stronger goals.

External attacks on the protocol obtain some of the messages through another run of the protocol. This run can be further subdivided into interleavings and classic replay attacks.

Interleavings *require* the second (or third, fourth, . . .) run of the attack to be executed in parallel with the attacked run. In other words, some part of the protocol forces the intruder to start a new run with another agent, to produce an answer for certain queries. Interleaved attacks are also known as *parallel session* attacks.

Classic replays do not require the runs of the attack to be executed in parallel. This allows the messages of the run to be replayed later, when the honest agent who originally sent the messages is no longer participating in the attacked run.

This type of attack can break injectivity, as it makes it possible to reuse the thread executed by an agent in multiple runs, without the agent executing all of these threads. This means that there is a difference between the number of runs the attacked agent executes and the number of runs that are used to construct the messages in these runs.

3.1.2 Destination taxonomy

Next to the differentiation between internal and external attacks (and the corresponding subdivision), there is a differentiation on how the destination of messages is modified: the destination taxonomy. The two classes in this taxonomy are *deflection* and *straight replay* attacks.

Straight replay attacks do not modify the intended recipient. He receives the messages, but either the intruder changes the order of messages, or makes messages appear to have been sent by an agent different from the original origin.

For example, in the attack on the Needham-Schroeder protocol found in Example 2.11, the event $send(Alice \rightarrow Bob : \{\!|N_{Alice}, N_{Bob}|\!\}_{pk(Bob)})$ is replayed to the intended recipient, *Bob*. However, the sender is changed, and the event occurring in the run by *Bob* is actually $send(Eve \rightarrow Bob : \{\!|N_{Alice}, N_{Bob}|\!\}_{pk(Bob)})$.

Deflection attacks change the intended recipient of a message. The new target depends on the content of a message.

Deflection attacks break weak agreement, as the agent whose messages are reflected expects them to be delivered to an agent who does not actually receive them. This means that she does not agree on the identity of the actual recipient.

Reflection is a special case of deflection, in which the message is returned to the sender. This is the case for attacks misusing symmetric keys where a message $\{|m|\}_{k(i,r)}$ can be constructed both by i and by r.

After appending this taxonomy to the taxonomy of origin, Syverson claims to have obtained a full taxonomy, and finishes. We believe, however, that a third dimension is necessary: a *sender taxonomy*.

3.1.3 Sender taxonomy

The taxonomy as described above does not provide a classification of how the sender of messages is modified. We add this taxonomy as a third dimension to the structural classification.

Similar to the destination taxonomy, the sender taxonomy differentiates between whether the sender is modified or not. We call the different classes *straight forward* and *fake sender* attacks.

Straight forward attacks do not modify the sender. This means that either the attack changes the recipient, or it changes the content of the messages.

Fake sender attacks modify the occurrence of the original sender.

Reflection is again a special case of this modification. In this case, the original recipient becomes the new sender, and the original sender becomes the new recipient.

3.2 Formalization of the taxonomy

In this section, we will formalize the taxonomy presented in Section 3.1 as predicates on attacks. Especially, we will try to formulate the predicates on the set of runs, Rs alone.

In the rest of this section, A denotes the attack classified.

3.2.1 Taxonomy of origin

Definition 3.1 (Internal vs. external)**.** If an attack is internal, this means that there is only one run involved in the attack.

$$internal(A) \Leftrightarrow |A.Rs| = 1$$

An external attack is the exact opposite of an internal attack. For completeness, we will expand this definition here:

$$external(A) \Leftrightarrow |A.Rs| > 1$$

Note that we do not allow attacks without runs. This is correct, as an attack without any run would not have a honest agent participating, and therefore no target of the attack.

For external attacks, there are two possibilities. Either the attack *interleaves* the runs, or this is not required, implying the attack is a *replay attack*.

Interleaved runs *require* an attacker to use the second run to respond to messages in the first. Structurally, there should be at least one message in the attacked run partially constructed from a message in another run, and at least one message in the other run should be constructed from a message in the attacked run.

To capture that a message of one run occurs in another, we require a definition of modification. We shall not introduce that definition here, but informally formulate what it should be able to express here. To this end, we introduce the following notation, abstracting away from all other properties of the modification: $e_2 \rightsquigarrow e_1$, for two (instantiated) events e_1 and e_2. Intuitively, the meaning of this expression is: e_1 is constructed using the message sent in e_2. We will fully define a model of modification capable of expressing the $\rightsquigarrow$-relation in the next chapter.

Definition 3.2 (Interleaved vs. replay)**.**

$$\begin{aligned} &\mathit{interleaved}(A) \Leftrightarrow \mathit{external}(A) \wedge (\exists r_i, r_j \in A.\mathit{Rs} : r_i \neq r_j \wedge \\ &\qquad (\exists e_i \in r_i, e_j \in r_j : e_j \rightsquigarrow e_i) \wedge (\exists e_j \in r_j, e_i \in r_i : e_i \rightsquigarrow e_j)) \end{aligned}$$

If an external attack is not interleaved, we classify it as a replay attack:

$$\begin{aligned} &\mathit{replay}(A) \Leftrightarrow \mathit{external}(A) \wedge (\forall r_i, r_j \in A.\mathit{Rs} : r_i \neq r_j \Rightarrow \\ &\qquad (\forall e_i \in r_i : \neg(\exists e_j \in r_j : e_j \rightsquigarrow e_i)) \vee \\ &\qquad\qquad (\forall e_j \in r_j : \neg(\exists e_i \in r_i : e_i \rightsquigarrow e_j))) \end{aligned}$$

3.2.2 Destination taxonomy

The destination taxonomy is captured by three predicates. These predicates again use the $\rightsquigarrow$-relation.

When the events of a run r_2 are replayed in run r_1 in a straight replay, the receiver of the events does not change.

Definition 3.3 (Straight replay)**.**

$$\begin{aligned} &\mathit{straight}(A, r_1, r_2) \Leftrightarrow r_1, r_2 \in A.\mathit{Rs} \wedge (\forall e_1 \in r_1, e_2 \in r_2 : \\ &\qquad e_2 \rightsquigarrow e_1 \wedge e_1 \equiv \mathit{read}(a_1 \rightarrow a_2 : m) \wedge e_2 \equiv \mathit{send}(a_1' \rightarrow a_2' : m') \Rightarrow \\ &\qquad\qquad a_2 = a_2') \end{aligned}$$

When the events of one run are deflected towards another run, the intended recipient does change.

Definition 3.4 (Deflection).

$$\begin{aligned}&\mathit{deflect}(A, r_1, r_2) \Leftrightarrow r_1, r_2 \in A.Rs \wedge (\forall e_1 \in r_1, e_2 \in r_2 :\\ &\qquad e_2 \rightsquigarrow e_1 \wedge e_1 \equiv \mathit{read}(a_1 \to a_2 : m) \wedge e_2 \equiv \mathit{send}(a_1' \to a_2' : m') \Rightarrow\\ &\qquad\qquad a_2 \neq a_2')\end{aligned}$$

Reflection is a special case of deflection.

Definition 3.5 (Reflection).

$$\begin{aligned}&\mathit{reflect}(A, r_1, r_2) \Leftrightarrow r_1, r_2 \in A.Rs \wedge (\forall e_1 \in r_1, e_2 \in r_2 :\\ &\qquad e_2 \rightsquigarrow e_1 \wedge e_1 \equiv \mathit{read}(a_1 \to a_2 : m) \wedge e_2 \equiv \mathit{send}(a_1' \to a_2' : m') \Rightarrow\\ &\qquad\qquad a_2' = a_1 \wedge a_1' = a_2)\end{aligned}$$

3.2.3 Sender taxonomy

The sender taxonomy follows the same ideas as the destination taxonomy, but now with the restrictions imposed on the sender of the messages.

Definition 3.6 (Straight forward).

$$\begin{aligned}&\mathit{forward}(A, r_1, r_2) \Leftrightarrow r_1, r_2 \in A.Rs \wedge (\forall e_1 \in r_1, e_2 \in r_2 :\\ &\qquad e_2 \rightsquigarrow e_1 \wedge e_1 \equiv \mathit{read}(a_1 \to a_2 : m) \wedge e_2 \equiv \mathit{send}(a_1' \to a_2' : m') \Rightarrow\\ &\qquad\qquad a_1 = a_1')\end{aligned}$$

Definition 3.7 (Fake sender).

$$\begin{aligned}&\mathit{fake}(A, r_1, r_2) \Leftrightarrow r_1, r_2 \in A.Rs \wedge (\forall e_1 \in r_1, e_2 \in r_2 :\\ &\qquad e_2 \rightsquigarrow e_1 \wedge e_1 \equiv \mathit{read}(a_1 \to a_2 : m) \wedge e_2 \equiv \mathit{send}(a_1' \to a_2' : m') \Rightarrow\\ &\qquad\qquad a_1 \neq a_1')\end{aligned}$$

Definition 3.8 (Reflection).

$$\begin{aligned}&\mathit{reflect}(A, r_1, r_2) \Leftrightarrow r_1, r_2 \in A.Rs \wedge (\forall e_1 \in r_1, e_2 \in r_2 :\\ &\qquad e_2 \rightsquigarrow e_1 \wedge e_1 \equiv \mathit{read}(a_1 \to a_2 : m) \wedge e_2 \equiv \mathit{send}(a_1' \to a_2' : m') \Rightarrow\\ &\qquad\qquad a_2' = a_1 \wedge a_1' = a_2)\end{aligned}$$

Note that reflection is both part of the sender taxonomy and of the destination taxonomy.

In this chapter, we have shown and formalized a taxonomy for structurally classifying attacks. This taxonomy is already capable of distinguishing between internal attacks (attacks breaking aliveness) and external attacks (attacks needing multiple runs), as well as describing how the runs of an external attack interrelate. Furthermore, the taxonomy describes how the role instantiation of runs are adapted between sending and reading.

What is missing is a more sophisticated look at *how* the attacker modifies the messages between the runs. So, we need to introduce a model for modifications, and a taxonomy of this model. This model and its taxonomy will be described in the next chapter.

Chapter 4

Modifications

The structural properties described in the previous chapter can together be considered as the skeleton of an attack. In particular, that taxonomy describes different ways an attacker can combine the runs of honest agents to achieve her goals.

On the other hand, the modifications, described by $A.\mathcal{G}$, can be considered as the muscles of an attack: they describe how the events of one run contribute to events in another run. Next to describing the relation between honest events, the modifications are extended with the actions of the intruder.

This chapter will investigate two possible formalisms for modifications. Although we have already hinted that we will choose the graph formalism for the modification, we will elaborate on this decision here.

4.1 Requirements for modification

Our goal is to describe the actions of the intruder in such a way that we can group certain combinations of actions into intuitive classes.

We impose some requirements and restrictions on the modification mechanism, namely:

Expressing $\leadsto$ The first requirement follows from the previous chapter, where we needed the relation $\leadsto$ to express some structural properties. This means that the modification mechanism should allow one to express this relation.

Specifically, the modification should allow the expression of the relation $\leadsto \subseteq \mathit{SendEvent} \times \mathit{ReadEvent}$, where $\mathit{SendEvent} = \{e : \mathit{Event} | (\exists A_1, A_2 : \mathit{Agent}, m : \mathit{Term} \cdot e \equiv \mathit{send}(A_1 \rightarrow A_2 : m))\}$ and $\mathit{ReadEvent} = \{e : \mathit{Event} | (\exists A_1, A_2 : \mathit{Agent}, m : \mathit{Term} \cdot e \equiv \mathit{read}(A_1 \rightarrow A_2 : m))\}$.

This requirement intuitively means that an intruder reads the term in a *send* event in some run, and uses it to construct a term occurring in a *read* event in some run.

Consistent with Dolev-Yao rules The modification should consist of actions consistent with the rules outlined in section 2.2.

Describe interaction of runs This requirement is the most difficult to grasp formally. Intuitively, we would like the modification to describe the way the messages occurring in the honest runs are related. This should describe what part each run plays in the entire attack.

4.2 Modifications modelled as functions

As a first attempt, we try to model the modification by using functions mapping events from one run to the events of another run. We can describe the modifications by a function $f_{i,j} : Event_i \rightarrow Event_j$, where $Event_k = \{e : Event | (\exists th \in r_k \cdot (\exists e_1 \in th.T \cdot e = th.\nu(e_1)))\}$.

There are two possible interpretations for $f_{i,j}$:

- $f_{i,j}$ takes as an argument a *send* event from run r_i and produces the *read* event in run r_j that is produced from this message,
- $f_{i,j}$ takes as an argument a *read* event in run r_i and produces the *send* event that was used to produce this event.

We shall consider both of these possibilities.

4.2.1 Sent to received

The first possibility describes how each sent message in run r_i is used by the intruder to construct the messages read by the honest agent executing run r_j.

The function $f_{i,j}$ can be partial: if a send event is not reused, the function does not map it to an event in r_j.

Example 4.1 (Needham-Schroeder modifications)**.** As an example, consider the attack on the Needham-Schroeder public key protocol. In this attack, there are two runs. Run one is carried out by the initiator *Alice* and the untrusted agent *Eve* as responder. The messages from this run are forwarded to run two, carried out by responder *Bob*. The attacker's goal is to make *Bob* believe he is talking to *Alice*.

In run one, the following events are created by *Alice*:

- $send(Alice \rightarrow Eve : \{\!|Alice, N_{Alice}|\!\}_{pk(Eve)})$
- $send(Alice \rightarrow Eve : \{\!|N_{Bob}|\!\}_{pk(Eve)})$

These are modified by $f_{1,2}$ into the following events received by *Bob*, respectively:

- $read(Alice \rightarrow Bob : \{\!|Alice, N_{Alice}|\!\}_{pk(Bob)})$
- $read(Alice \rightarrow Bob : \{\!|N_{Bob}|\!\}_{pk(Bob)})$

In run one, the following event is created by *Bob*:

- $send(Bob \rightarrow Alice : \{\!|N_{Alice}, N_{Bob}|\!\}_{pk(Alice)})$

Which is modified by $f_{2,1}$ into the following event received by *Alice*:

- $read(Eve \rightarrow Alice : \{\!|N_{Alice}, N_{Bob}|\!\}_{pk(Alice)})$

So, only the sender is replaced by *Eve*.

This version of modification functions has a few difficulties apart from the general problems of functional modification, which we will describe in Section 4.2.3.

First, if an event from run r_i is used multiple times, for example when a leaked key is used to create multiple encrypted messages, this cannot be expressed by a purely functional application.

Furthermore, it is possible that a *read* event in r_j was not constructed using an event from r_i. Although this is not a problem in a theoretical sense, it is desirable to identify messages constructed purely from attacker knowledge, as these point to messages in the protocol which are unused or under control of the attacker.

This could be remedied by extending the domain of $f_{i,j}$ with a single event *knowledge*, representing the intruder reading initial knowledge. However, this still suffers from the first problem of using these functions, since the initial knowledge can be reused multiple times. The initial knowledge gets reused even more times than individual messages. In Example 4.1 alone, we obtain $pk(Bob)$ from the initial knowledge multiple times.

4.2.2 Received to sent

The problems mentioned in the previous section could be solved by turning the function around, defining a function which takes as an argument a read event in r_i, and produces the send event from r_j contributing to this event.

This approach does not suffer from the second problem described in the previous section (read messages constructed entirely from initial knowledge), as we can use the special element *knowledge* as a result to show that an event is not constructed from any event in r_j. If there are events in r_i for which $f_{i,j}$ produces *knowledge* for each run r_j of the attack, then these events are made from the initial attacker knowledge.

However, with the function defined as such, we cannot account for messages constructed from more than one message.

4.2.3 Final thoughts on modification functions

From the previous two sections, we can conclude that modification cannot be easily captured by a functional notation: multiple messages can be used to construct one message, and a message can be used multiple times for constructing other messages.

So, instead of a functional definition, we need to generalize the modification as a relation between messages.

Having established modifications as a relation, we still cannot classify on the actions taken by the attacker, as these are not shown by relating two messages. To show the constructive nature of the attack, we build a graph for these actions.

4.3 Modifications modelled as graphs

We can build a directed, acyclic graph for each *read* event by an honest agent, by following the attacker's actions leading up to this *read*.

These actions are represented by a set $\mathcal{A}$ consisting of the following actions:

Intercepting a term sent by an honest agent.

Obtaining a term from the initial intruder knowledge.

Encrypting a term with another term. Encryption also includes signing a message, in a public key infrastructure.

Decrypting a term $\{\!|t_1|\!\}_{t_2}$ with term t_2^{-1}.

Tupling two terms into a new term.

Untupling a tuple term (t_1, t_2) to obtain either t_1 or t_2.

Applying a function term to an argument term, to obtain a new term. We assume that obtaining the argument of a function application $f(t)$ is only possible by applying the inverse function f^{-1} to the application, so $f^{-1}(f(t)) = t$

In this set, we do not consider the actual sending of the message to the receiving agent, so there is no transition labelled "send".

The nodes of the graph represent the subterms in the attacker knowledge, used in constructing the term read.

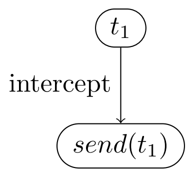
4.3.1 Graph construction

We construct a graph for each read event of each honest agent, by placing the term read in this event in the initial node.

We extend the graph by adding outgoing edges to each node. The edges point to the subterms used in creating the term, and the edges represent the action the intruder takes to combine terms. This means that for encryption, decryption, tupling, and functional application, there are two possible choices for the attacker actions: either the term was obtained as is, or it was constructed by the attacker from the subterms.

Based on the possible actions in $\mathcal{A}$ and the observation that construction of a term has two operands, we get the following possible edges:

Intercept The target node is a send by an honest agent, directly forwarded to the source node.



Obtain The target node is the node representing the initial knowledge of the attacker.

$$\begin{tikzcd}
t_1 \arrow[d, "\text{obtain}"'] \\
\mathit{knowledge}(t_1)
\end{tikzcd}$$

In the above graph, $\mathit{knowledge}(t_1)$ represents the set of initial knowledge $\mathcal{K}$, such that $t_1 \in \mathcal{K}$.

Encryption Is a composite operation with two arguments: the plaintext to be encrypted, and the key with which the encryption is carried out.

Encrypt-plain The target node is the plaintext encrypted within the source node.

Encrypt-key The target node is a key used for creating the encryption in the source node.

$$\begin{tikzcd}
& \{\!|t_1|\!\}_{t_2} \arrow[dl, "\text{encrypt-plain}"'] \arrow[dr, "\text{encrypt-key}"] & \\
t_1 & & t_2
\end{tikzcd}$$

Decryption Is a composite operation with two arguments: the encrypted term that needs to be decrypted, and the key used in decryption.

Decrypt-cipher The target node is the ciphertext that is decrypted in order to obtain the source node.

Decrypt-key The target node is the key used to obtain the plaintext in the source node.

$$\begin{tikzcd}
& t_1 \arrow[dl, "\text{decrypt-cipher}"'] \arrow[dr, "\text{decrypt-key}"] & \\
\{\!|t_1|\!\}_{t_2} & & t_2^{-1}
\end{tikzcd}$$

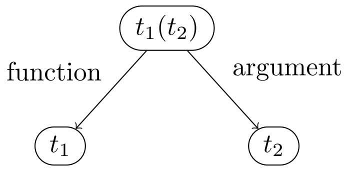
Tuple The target node is one of the operands of the tuple of the source node.

$$\begin{tikzcd}
& (t_1, t_2) \arrow[dl, "\text{tuple}"'] \arrow[dr, "\text{tuple}"] & \\
t_1 & & t_2
\end{tikzcd}$$

Untuple The target node is a tuple containing the source node

$$\begin{tikzcd}
t_1 \arrow[dr, "\text{untuple}"'] & & t_2 \arrow[dl, "\text{untuple}"] \\
& (t_1, t_2) &
\end{tikzcd}$$

Function & argument The source node is the result of applying a function. The function and its argument are in the target nodes.



These construction rules seem to accommodate a non-deterministic choice at certain points, if the attacker has a choice between constructing a composite term or intercepting this term when it is sent by an honest agent. Because we are analyzing the attack a posteriori, however, we have assume to be able to determine how the attacker obtained a result from the attack's trace.

In Scyther, for example, an attack is given by a number of traces, each consisting of one or more read events, and one send event. The send event contains the term constructed by the intruder, using the read events preceding it. If one of these read events correspond to the send of an honest agent, this trace corresponds to an intercept, if the read events only correspond to a send of the intruder, it corresponds to a construction.

So, the trace

$$read(* : pk(Alice)) \leq read(* : m) \leq send(Eve \rightarrow Alice : \{\!|m|\!\}_{pk(Alice)})$$

represents an encryption by the intruder. In the read events of this trace, the $*$ instead of the source and target represents that we do not care where these events were obtained. On the other hand, the trace

$$read(Bob \rightarrow Alice : \{\!|m|\!\}_{pk(Alice)}) \leq send(Eve \rightarrow Alice : \{\!|m|\!\}_{pk(Alice)})$$

represents that an intercept was used to obtain the same term.

In constructing the graph, we observe four distinct kinds of nodes:

- An initial node, containing the term read.
- A final node depicting that an event originates from the initial intruder knowledge.
- A final node depicting that an event was sent by an honest agent.
- Intermediate nodes depicting the (de)construction of terms.

Of special interest is the fact that there are two kinds of final nodes. One of these represents that the attacker already knew the terms leading to it, the other representing the intruder using a message sent in a run in her attack. This difference indicates that a read term has three different possible origins:

- The term was constructed by using one or more terms send by honest agents.
- The term was constructed entirely from the attacker's initial knowledge.
- The term was constructed using a mix of both.

The graphs constructed by this method represent the way the intruder constructed the terms delivered to the honest agent, as well as how the knowledge obtained by the attacker, either initially or through reading a message, is used. As such, it depicts the modification relation between the events of the different runs in the attack.

Together, all graphs for the read events of honest agents in the attack A form the set $A.\mathcal{G}$.

4.3.2 Expressing $\leadsto$

We can now easily express the $\leadsto$ relation, given the set $A.\mathcal{G}$.

Definition 4.2 ($\leadsto$ on graphs)**.** Two events e_1 and e_2 are related by the $\leadsto$-relation, notation $e_1 \leadsto e_2$, iff:

- $e_1 \equiv \mathit{send}(A_1 \rightarrow A_2 : m)$, for some agents A_1, A_2 and message m.
- $e_2 \equiv \mathit{read}(A'_1 \rightarrow A'_2 : m')$, for some agents A'_1, A'_2 and message m'.
- There is a $g \in A.\mathcal{G}$, such that m' is the initial node of g, and m is one of the final nodes of g.

4.4 Equivalence of modifications

When comparing the modifications of two attacks, we are not interested in most of the subterms, so we disregard the node labels. Instead, we consider the nodes equivalent if they have the same paths to initial knowledge or honest messages.

Definition 4.3 (Equivalence of term nodes)**.** Two term nodes n_1 and n_2 are equivalent, notation $n_1 \leftrightarrow n_2$ iff:

- n_1 and n_2 are both final nodes representing the initial attacker knowledge, or
- n_1 and n_2 are both final nodes representing a term sent by a honest agent, or
- for each edge $n_1 \xrightarrow{a} n'_1$, there is an edge $n_2 \xrightarrow{a} n'_2$, such that $n'_1 \leftrightarrow n'_2$, and for each edge $n_2 \xrightarrow{b} n'_2$, there is an edge $n_1 \xrightarrow{b} n'_1$, such that $n'_2 \leftrightarrow n'_1$.

This relation is similar to the bisimulation relation from process algebra, with the exception that we have two different "final" nodes here, which also need to be related.

Similarly to bisimulation, we define the equivalence on the total graphs, by requiring that their initial nodes are related.

Definition 4.4 (Equivalence of graphs)**.** Two graphs g_1 and g_2 are equivalent, if there is an equivalence $\leftrightarrow$ such that for the initial nodes i_1 of g_1 and i_2 of g_2, $i_1 \leftrightarrow i_2$.

We overload the notation $\leftrightarrow$ to also apply to the equivalence of graphs.

4.5 Classifying modifications

In theory, there is an infinite number of possible action combinations taken by the attacker, as there is no limit on the ways a term can be recombined in tuples, functional applications and encryptions.

We can reduce the number of combinations by requiring any attack to keep the path from a term to its smallest subterms as short as possible.

By this we mean that a term is not encrypted and decrypted, or tupled and untupled infinitely many times. It does *not* mean that when a path given by the attack leads to the initial knowledge, while there exists a path with less steps to a honest send, the second path is preferred in constructing the graph. Instead, these two paths represent two different modifications.

If an attack graph is not minimal as described above, it can be minimized by removing redundant edges and nodes. For our classification, we therefore assume that all attack graphs are minimal.

We shall now describe our taxonomy of modification graphs. A classification of this taxonomy describes how the initial node of a graph was obtained by the intruder: either she constructed it from smaller subterms, or it was obtained as a whole at some point during the protocol run.

In the formal definition of the classes, we assume the following functions on graphs are available:

$init(g)$ Should return the initial node of a graph g.

$final(g)$ Should return the set of final nodes of a graph g.

$nodes(g)$ Should return all nodes occurring in the graph g.

$honest(n)$ Should be a predicate telling if the node n represents a honest send event.

4.5.1 Origin of terms

This taxonomy is based on the origin of the term occurring in event e. As we already noted, there are three possible origins for a term: it can be either fully intercepted, fully fabricated out of the initial knowledge, or a combination of the two sources.

Definition 4.5 (Full intercept)**.** A graph g is a full intercept modification for an event e if it fulfils the predicate $full\text{-}intercept(e, g)$:

$$\begin{aligned} &full\text{-}intercept(e, g) \Leftrightarrow \\ &\qquad e = init(g) \wedge (\forall n \in final(g), n' \in nodes(g) \cdot n' \xrightarrow{l} n \Rightarrow l = intercept) \end{aligned}$$

Definition 4.6 (Full fabrication)**.** A graph g is a full fabrication modification for an event e if it satisfies the predicate $full\text{-}fabrication$, defined by:

$$\begin{aligned} &full\text{-}fabrication(e, g) \Leftrightarrow \\ &\qquad e = init(g) \wedge (\forall n \in final(g, n' \in nodes(g) \cdot n' \xrightarrow{l} n \Rightarrow l = obtain) \end{aligned}$$

When a modification is neither a full intercept, nor a full fabrication, we call it a partial intercept (notation: *intercept*), since we want to focus on how the messages sent by honest agents are reused.

Definition 4.7 (Intercept)**.** A graph g is an intercept modification for an event e if it neither satisfies $full\text{-}intercept$, nor $full\text{-}fabrication$:

$$\begin{aligned} &intercept(e,g) \Leftrightarrow \\ &\qquad e = init(g) \wedge \neg(full\text{-}intercept(e,g) \vee full\text{-}fabrication(e,g)) \end{aligned}$$

An intercept or full intercept modification can be further classified based on the number of distinct send events used for the modification: either all term are obtained from one honest send, or several honest sends are used to construct the terms.

Definition 4.8 (Single point of origin)**.** A graph g has a single point of origin for event e, if it satisfies:

$$single\text{-}origin(e,g) \Leftrightarrow e = init(g) \wedge |\{n \in final(g) | honest(n)\}| = 1$$

Definition 4.9 (Combining modification)**.** A graph g is a combining modification for event e, if it satisfies:

$$combination(e,g) \Leftrightarrow e = init(g) \wedge |\{n \in final(g) | honest(n)\}| > 1$$

For reparation purposes, the modifications that is a full intercept with a single point of origin is important. We will call this class the class of forwarding modifications.

Definition 4.10 (Forwarding modification)**.** A graph g is a forwarding modification for an event e if it satisfies:

$$mforward(e,g) \Leftrightarrow e = init(g) \wedge full\text{-}intercept(e,g) \wedge single\text{-}origin(e,g)$$

In this chapter, we have shown a model of message modifications based on modification graphs. These graphs represent the actions an attacker has taken before delivering a certain message to an honest agent.

Once we have constructed the modification graphs, we can determine when two modifications are equivalent, disregarding the actual messages they manipulate. This notion of equivalence is based on the choices made by the attacker, and is similar to the notion of bisimulation found in process algebra.

We can also classify a graph according to a taxonomy based on its structure. This classification is especially useful to determine which messages have been taken from other runs.

What we will show in the next chapter is how we can use our model and classifications to analyze attacks against a protocol. This analysis is based on the classification of an attack within the different taxonomies, supplemented by reasoning on a more abstract level.

Chapter 5

Using Classification in Attack Analysis

In this chapter, we shall show how the taxonomy described in the previous two chapters can be used in analyzing attacks.

We will first look at the class of man-in-the-middle attacks against challenge-response mechanisms, showing how several attacks against different protocols are actually similar, and how these similarities show in the classification of the attack.

Next, we shall show how the classification works for analyzing different attacks against a single protocol, illustrated by an example analysis of attacks against the TMN protocol.

Finally, we shall show how the classification of an attack can be used in finding a flaw in the protocol, and repairing the protocol based on the attack. We illustrate this by looking at the Woo and Lam Π protocol.

5.1 Classifying authentication attacks

5.1.1 Man-in-the-middle attacks

A man-in-the-middle (MITM) attack is an attack that breaks a challenge-response mechanism in an authentication protocol.

This mechanism is a way to prove freshness, which is one of the two elements of authenticity, as was mentioned in Chapter 1. The basic exchange of challenge and response is shown in the message sequence chart in Figure 5.1

In this exchange, the challenger c sends out a challenge C. This challenge contains a nonce, and possibly the identity of either c, r, or both. This identity does not necessarily have to occur as an atomic notion in the message. Instead, it can also occur as a part of the key. For example, usage of a long-term, symmetric key shared between c and r provides the identities of both agents in one encrypted message $\{\!|n_c|\!\}_{k(c,r)}$. From the challenge, the responder r should extract enough information to prove his identity and being active. Furthermore, the responder should also be able to derive the identity of the challenger.

The proof of identity can be included in a similar way as in the challenge, either by using the appropriate keys or by including the name in the message.

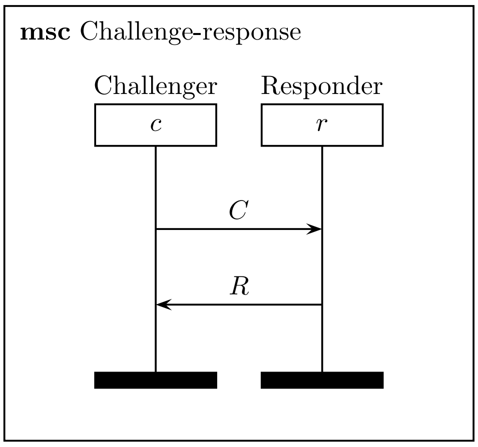


Figure 5.1: Challenge-response message exchange

The freshness is proved by applying one or more operations on the nonce. This can be decrypting the message containing the nonce and encrypting the nonce under a different key, or applying a function to the nonce and returning it.

Upon receiving the response R, the challenger checks that it corresponds to the nonce sent, and that it was constructed by r. If this is the case, c can accept that r is alive and communicating with her. In other words, c *authenticates* r.

Instantiations of the challenge-response pattern

The pattern can be instantiated in different ways, depending on what operations are chosen to construct challenges and responses. Furthermore, an instantiated pattern can be embedded within a larger protocol, for example to establish mutual authentication.

We have already seen a challenge-response instantiation in the Needham-Schroeder protocol of Example 2.5. The message sequence chart of that example is repeated in Figure 5.2.

This protocol actually uses two challenge-response pairs:

- Challenge $C_1 = \{|i, n_i|\}_{pk(r)}$ paired with response $R_1 = \{|n_i, n_r|\}_{pk(i)}$.
- Challenge $C_2 = \{|n_i, n_r|\}_{pk(i)}$ paired with response $R_2 = \{|n_r|\}_{pk(r)}$.

So, the message $\{|n_i, n_r|\}_{pk(r)}$ serves both as the challenge C_2 and as the response R_1.

The pair (C_1, R_1) proves that responder r has received the challenge C_1, by decrypting the message, removing i from it, and appending another nonce n_r to it, before encrypting the new message again. As only r can manipulate a message like this, and nonce n_i is fresh, the pair is enough to prove to initiator i that r is active and communicating with her.

The second pair intends to prove to r that i is active and communicating with him. This proof is given by i decrypting the challenge, and stripping n_i from the resulting tuple. Afterwards, she encrypts the nonce n_r. Only i can decrypt the challenge, and the nonce provides the freshness proof.

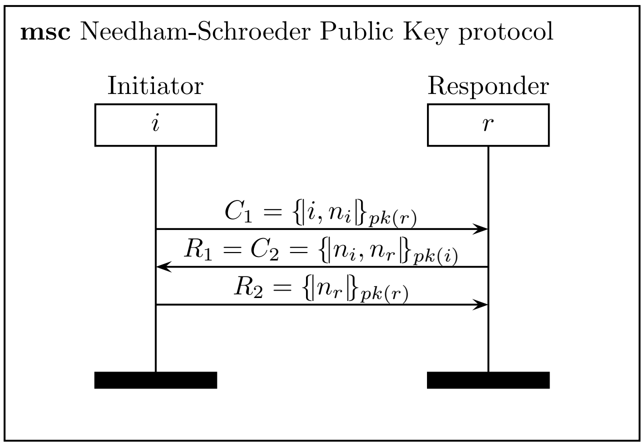

msc Needham-Schroeder Public Key protocol
Initiator
i
Responder
r
$C_1 = \{|i, n_i|\}_{pk(r)}$
$R_1 = C_2 = \{|n_i, n_r|\}_{pk(i)}$
$R_2 = \{|n_r|\}_{pk(r)}$

Figure 5.2: Needham-Schroeder public key protocol MSC

Nevertheless, the protocol is susceptible to an attack, as is shown in Example 2.11. We will analyze this attack in depth later in this chapter.

Another instantiation of the challenge-response pattern can be found in one of the SPLICE/AS protocols [3], which is shown in the MSC in Figure 5.3.

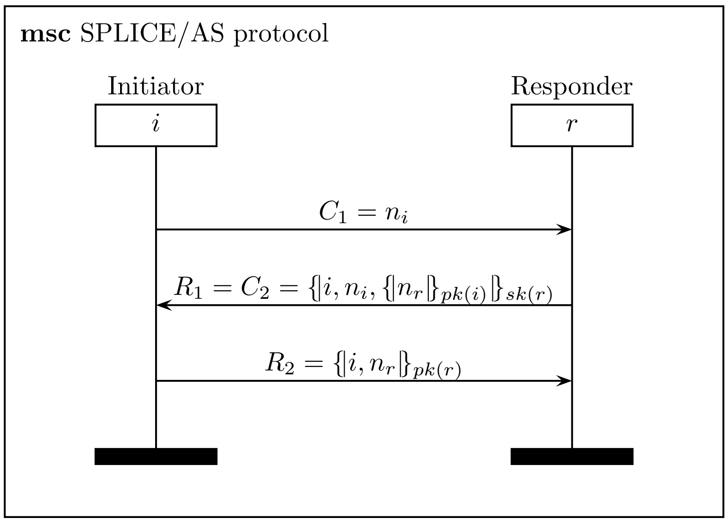

msc SPLICE/AS protocol
Initiator
i
Responder
r
$C_1 = n_i$
$R_1 = C_2 = \{|i, n_i, \{|n_r|\}_{pk(i)}|\}_{sk(r)}$
$R_2 = \{|i, n_r|\}_{pk(r)}$

Figure 5.3: The SPLICE/AS protocol

This is again a mutual authentication protocol, using two challenges and two responses.

The challenge C_1 from challenger i to responder r is an unencrypted nonce n_i, used to prove freshness.

The nonce returns to i inside the response $R_1 = \{|i, n_i, \{|n_r|\}_{pk(i)}|\}_{sk(r)}$, so it is signed with the secret key of r. This is an operation only r could have carried out, proving that r participated in the current run of the protocol.

The response R_1 doubles as challenge C_2, containing the element $\{|n_r|\}_{pk(i)}$. This is a message only the responder i can read, and she responds by returning $R_2 = \{|i, n_r|\}_{pk(r)}$.

This protocol is also vulnerable to an attack.

A third protocol using this pattern is the CCITT X.509 3 messages protocol [4]. We will not reiterate this protocol here, but it is similar to the NSPK protocol, with the difference that the challenges and responses are this time enhanced by digital signatures instead of encryption, and that there is a message preceding the challenge-response exchange.

Attacks on challenge-response

The best known attack against a challenge-response mechanism is the attack Lowe mounted against the NSPK protocol. The MSC of this attack is repeated in Figure 5.4.

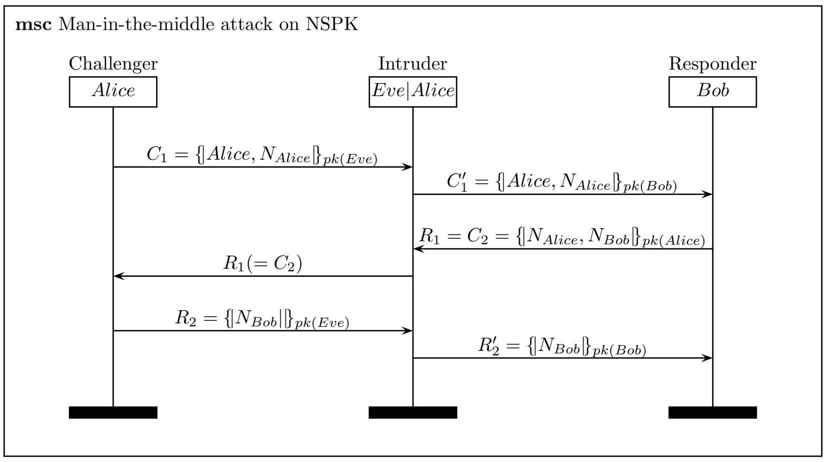

msc Man-in-the-middle attack on NSPK
Challenger
Alice
Intruder
Eve|*Alice*
Responder
Bob
$C_1 = \{|Alice, N_{Alice}|\}_{pk(Eve)}$
$C'_1 = \{|Alice, N_{Alice}|\}_{pk(Bob)}$
$R_1 = C_2 = \{|N_{Alice}, N_{Bob}|\}_{pk(Alice)}$
$R_1(= C_2)$
$R_2 = \{|N_{Bob}|\}_{pk(Eve)}$
$R'_2 = \{|N_{Bob}|\}_{pk(Bob)}$

Figure 5.4: Lowe's man-in-the-middle attack on the NSPK protocol

The attack abuses the fact that C_2 does not contain a mention of the original sender, *Bob*. This means that the intruder can send it as any other message C_2. In particular, the message is forwarded to *Alice*, as if it was constructed by *Eve*.

Alice replies to C_2 with R_2, which is encrypted under *Eve*'s public key. The intruder then re-encrypts R_2, constructing R'_2, and sends it to *Bob*. Upon receiving R'_2, *Bob* thinks it was constructed by *Alice* and falsely authenticates her.

Classifying the attack

The attack of Lowe can be classified in our framework. We will first show the structural classification, and continue with the modification graphs and the classification thereof.

As we have already shown in Example 2.11, the attack contains two runs: run one, between *Alice* and *Eve*, in which the messages C_1, R_1 and R_2 are exchanged, and run two, between *Bob* and the intruder, in which the messages C'_1,

R_1 and R_2' are exchanged. So, this attack is an *external* attack. In particular, the attack contains both messages in run one that are forwarded to the second run, namely $C_1 \rightsquigarrow C_1'$ and $R_2 \rightsquigarrow R_2'$, and one message that is returned from the second run to the first run: $R_1 \rightsquigarrow R_1$. This implies that the attack is an *interleaved* attack with respect to the origin taxonomy.

With respect to the destination taxonomy, the attack is a *deflection* attack for run one: all messages of this run are sent to *Bob*, while they were intended for *Eve*. The message from the second run is forwarded in a *straight* manner: R_2 was intended for *Alice* and is actually delivered to her.

The sender classification is the inverse of the destination classification: messages from run one to run two retain their original sender (*Alice*), while the message from run two back to run one replace *Bob* with *Eve*.

For the modifications, we shall give the graphs for each of the three messages read by the honest agents.

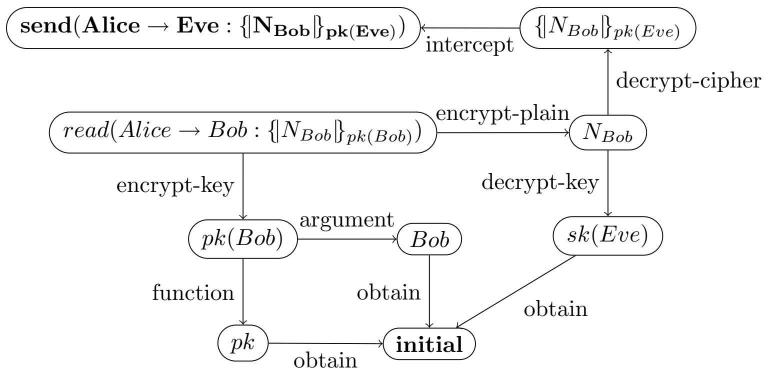
$read(Alice \rightarrow Bob : C_1)$ This graph is fairly complex, but entails re-encrypting C_1 under *Bob*'s public key. The two final states are in bold face for easy recognizability.

$$\begin{tikzcd}
\mathbf{send(Alice \rightarrow Eve : \{\!|Alice, N_{Alice}|\!\}_{pk(Eve)})} & & & \{\!|Alice, N_{Alice}|\!\}_{pk(Eve)} \arrow[lll, "intercept"'] \\
read(Alice \rightarrow Bob : \{\!|Alice, N_{Alice}|\!\}_{pk(Bob)}) \arrow[rrr, "encrypt-plain"] \arrow[dr, "encrypt-key"'] & & & Alice, N_{Alice} \arrow[u, "decrypt-cipher"'] \arrow[d, "decrypt-key"'] \\
& pk(Bob) \arrow[r, "argument"] \arrow[d, "function"'] & Bob \arrow[d, "obtain"'] & sk(Eve) \arrow[dl, "obtain"] \\
& pk \arrow[r, "obtain"'] & \mathbf{initial} &
\end{tikzcd}$$

$read(Eve \rightarrow Alice : R_1)$ This graph displays the intruder forwarding *Bob*'s challenge to *Alice*. It is a simple intercept modification, representing a forward of the message from one run to another.

$$\begin{tikzcd}
read(Eve \rightarrow Alice : \{\!|N_{Alice}, N_{Bob}|\!\}_{pk(Alice)}) \arrow[d, "intercept"'] \\
send(Bob \rightarrow Alice : \{\!|N_{Alice}, N_{Bob}|\!\}_{pk(Alice)})
\end{tikzcd}$$

$read(Alice \rightarrow Bob : R_2')$ The modification for this event is structurally equivalent to the modification of C_1. We will repeat the graph, but now with the subsets of the modification with R_2'.


$\mathbf{send(Alice \to Eve : \{\!|N_{Bob}|\!\}_{pk(Eve)})}$
intercept
$\{\!|N_{Bob}|\!\}_{pk(Eve)}$
decrypt-cipher
$read(Alice \to Bob : \{\!|N_{Bob}|\!\}_{pk(Bob)})$
encrypt-plain
N_{Bob}
encrypt-key
decrypt-key
$pk(Bob)$
argument
Bob
$sk(Eve)$
function
obtain
obtain
pk
obtain
initial

We can now classify the modifications described above.

The first and the last graph are *intercept* modifications, with a single point of origin. Because it also contains parts from the initial attacker knowledge, it is hard to put a more distinct classification on it, although it is interesting to note that the graphs represent the re-encryption of *Alice*'s messages.

A more interesting graph is the second graph (representing $R_1 \rightsquigarrow R_1$). This graph is a full-intercept modification, with a single point of origin. It is for this modification that Lowe's reparation fixes the protocol: because the message $\{\!|Bob, N_{Alice}, N_{Bob}|\!\}_{pk(Alice)}$ contains a reference to the original sender, and the intruder cannot modify the message (testified by the full intercept), the sender cannot be mistaken anymore within run one.

The attack on the SPLICE/AS protocols is shown in Figure 5.5. This attack is, like Lowe's attack, an interleaved attack, as there are two messages from the run between *Alice* and *Eve* forwarded to the run between *Alice* and *Bob*, and one message returned from the run between *Alice* and *Bob* to the run between *Alice* and *Eve*.

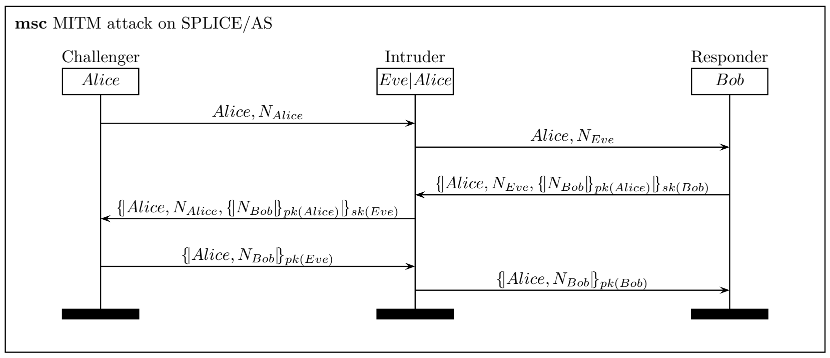

msc MITM attack on SPLICE/AS
Challenger
$Alice$
Intruder
$Eve|Alice$
Responder
Bob
$Alice, N_{Alice}$
$Alice, N_{Eve}$
$\{\!|Alice, N_{Eve}, \{\!|N_{Bob}|\!\}_{pk(Alice)}|\!\}_{sk(Bob)}$
$\{\!|Alice, N_{Alice}, \{\!|N_{Bob}|\!\}_{pk(Alice)}|\!\}_{sk(Eve)}$
$\{\!|Alice, N_{Bob}|\!\}_{pk(Eve)}$
$\{\!|Alice, N_{Bob}|\!\}_{pk(Bob)}$

Figure 5.5: Attack on SPLICE/AS

The modifications are classified differently. The first message is constructed entirely from the attacker's knowledge. In particular, the nonce N_{Eve} accepted by *Bob* is under control of the intruder. This is not necessarily a larger problem than the fact that authentication is broken, unless the nonce is used in a follow up to the protocol, for example as a random seed in key generation.

For the modification of the challenge-response pairs, the inner challenge $\{|N_{Bob}|\}_{pk(Alice)}$ is forwarded directly from the run $(Alice, Bob)$ to the run $(Alice, Eve)$. We can include the sender's identity in this message, and obtain a reparation similar to Lowe's fix for the NSPK protocol.

From these two examples we set the hypothesis that any attack that has two interleaved runs, and forwards a challenge directly from one run to the other, while forwarding the response using re-encryption, can be called *man-in-the-middle* attack. We have reinforced this hypothesis by running our prototype against the CCITT X.509 3 messages protocol, obtaining an equal classification for attacks against that protocol. The reparation can be obtained by adding the identity of the both sender and responder inside the encryption of the challenge being forwarded.

5.2 Multiple attacks against a single protocol

We shall show how our framework can be used in analyzing several attacks against a single protocol, the TMN protocol. The TMN protocol [20] uses a server to distribute a symmetric key from the initiator to the responder. The protocol description is given in Figure 5.6.

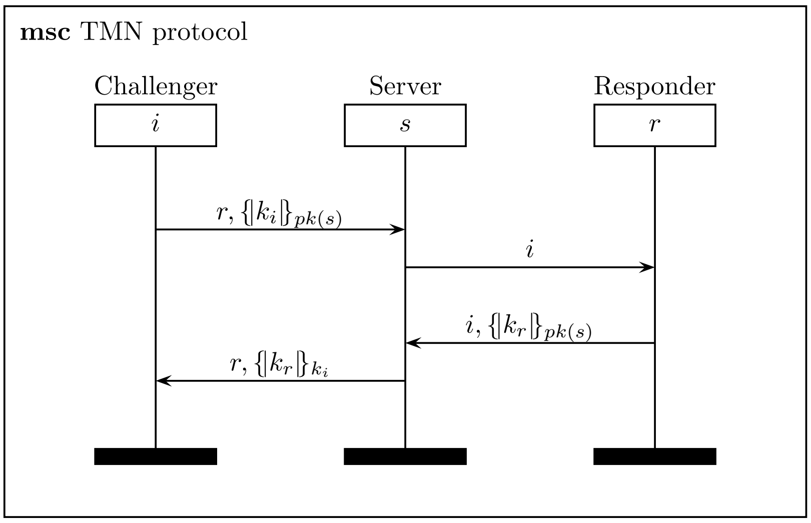


Figure 5.6: TMN protocol

In this protocol, the initiator first sends a session key (k_i) to the server, encrypted with its public key. The server consecutively tells the responder that the initiator wants to communicate with him. For this communication, the responder generates a new session key (k_r), which he sends to the server, again under its public key. The server, finally, encrypts the session key with the key it got from the initiator.

Because the key k_i was generated in this run, the initiator knows that the session was not replayed, and that she can use the key k_r for secure communication with the responder.

However, the responder can be tricked into believing the initiator wants to establish a session, which is shown in the internal attack of Figure 5.7. This attack was found by the Scyther tool.

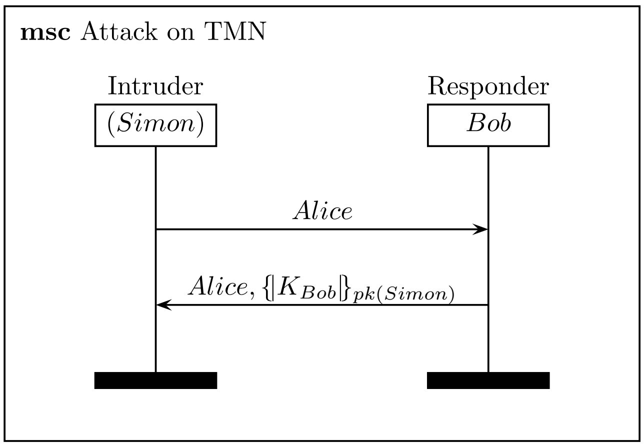


Figure 5.7: Attack on TMN found by Scyther

This attack shows that the intruder can trick the responder *Bob* into believing *Alice* wants to start talk to him. However, *Alice*, nor the server *Simon* actually participated in the protocol.

This attack does not seem to cause too much trouble, since the attacker cannot do anything with the message sent by *Bob*. There are, however, two attacks using this particular run. One, reported by Lowe and Roscoe [15], expands the run with a thread of the server, getting the attack of Figure 5.8.

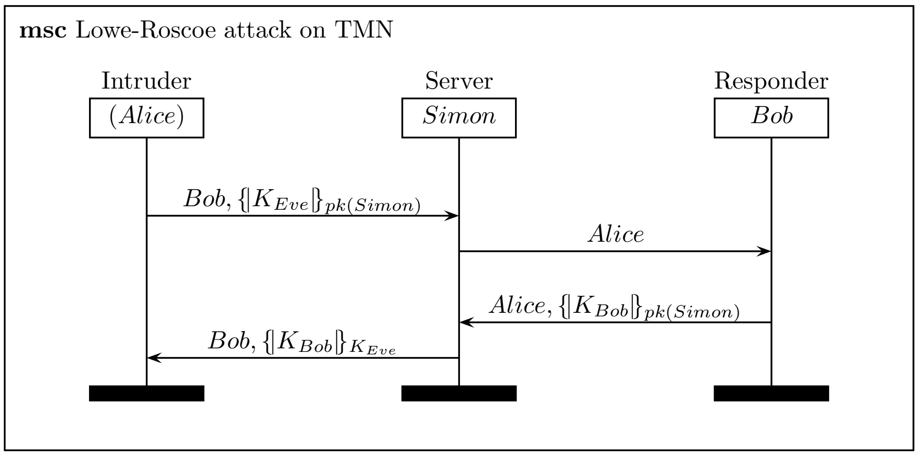


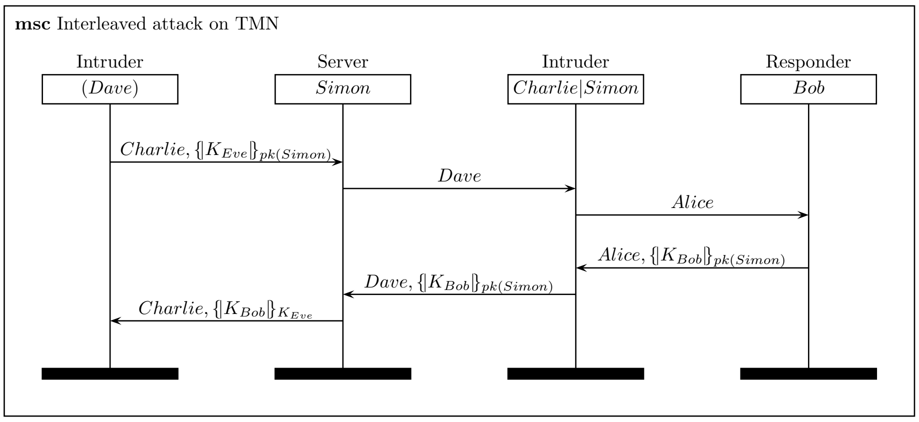
Figure 5.8: Internal attack on TMN found by Lowe and Roscoe

This attack is again an internal attack. The threads of the server, *Simon* and that of the responder *Bob* form a single run, of which the only further communication is with the intruder who inserts and intercepts messages.

This attack is far more destructive than that in Figure 5.7, as the intruder can read the key K_{Bob}, and use it to impersonate *Alice* in any session in which this key is used.

It seems impossible to repair this protocol, as there is no way for the server to tell that the initial message was actually sent by *Alice*. The only possibility would be by letting the initiator sign the complete initial message, but this breaks down the elegance of the protocol only requiring the server to have public and private keys.

The second attack is found by Scyther, and constructs a run between the server and an untrusted agent. It is depicted in Figure 5.9



msc Interleaved attack on TMN
Intruder
(*Dave*)
Server
Simon
Intruder
Charlie|*Simon*
Responder
Bob
$Charlie, \{|K_{Eve}|\}_{pk(Simon)}$
$Dave$
$Alice$
$Alice, \{|K_{Bob}|\}_{pk(Simon)}$
$Dave, \{|K_{Bob}|\}_{pk(Simon)}$
$Charlie, \{|K_{Bob}|\}_{K_{Eve}}$

Figure 5.9: External attack on TMN found by Scyther

This attack uses two runs. Run one is executed by the intruder pretending to be agent *Dave* and by the server *Simon*, while run two is executed by the responder *Bob*, who believes to be communicating with *Simon*, this second run is exactly the same as the run of *Bob* in the attack in Figure 5.7. What is especially interesting in this attack, is that the agent names *Dave* and *Charlie* could be replaced by any value, since neither of their names occurs in the second run. In particular, if we replace *Dave* with *Alice*, and *Charlie* with *Bob*, we get the runs of Figure 5.10.

The only modifications between the two runs in this figure represent the identity, mapping the message *Alice* to the message *Alice*, and the message $(Alice, \{|K_{Bob}|\}_{pk(Simon)})$ to the message $(Alice, \{|K_{Bob}|\}_{pk(Simon)})$. This means that between run one and run two, the intruder acts as a normal network node, and we can collapse the attack, obtaining Figure 5.8. This implies that the internal attack of Figure 5.8 is part of a larger class, defined by that of Figure 5.9. It also implies that the attack in Figure 5.9 cannot be repaired by a simple modification to one message, since the more specific, internal attack in Figure 5.8 cannot easily be repaired, as we already argued.

The reasoning above is an indication of how we can use our existing framework in reducing attacks, obtaining a simpler attack for which we can ascertain

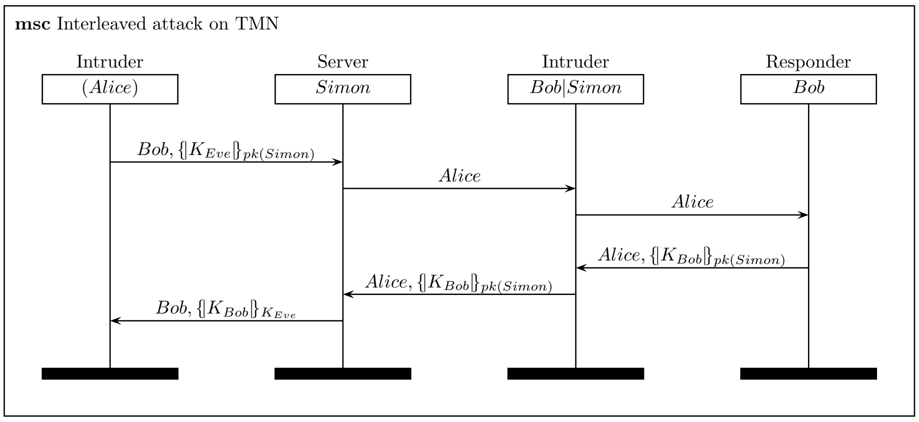

msc Interleaved attack on TMN
Intruder
(Alice)
Server
Simon
Intruder
Bob|Simon
Responder
Bob
$Bob, \{\!|K_{Eve}|\!\}_{pk(Simon)}$
$Alice$
$Alice$
$Alice, \{\!|K_{Bob}|\!\}_{pk(Simon)}$
$Alice, \{\!|K_{Bob}|\!\}_{pk(Simon)}$
$Bob, \{\!|K_{Bob}|\!\}_{K_{Eve}}$

Figure 5.10: External attack on TMN after substitution

the feasibility of attack, or generalize one attack to another, in order to obtain a reparation that works for both.

To automate this line of reasoning, we would need a more semantical classification of the modification, so that it would be capable of identifying which modifications are the identity. Also, the framework should capable of identifying those elements of runs that are not fixed to one agent name. Due to the time limitations of the master's project, we could not extend our framework to support these notions, but it would be an interesting avenue for further research.

5.3 Repairing protocols

When we have found the attacks against a protocol, we can use the classification of each attack to help determine what is wrong with the protocol, and how to repair the identified flaws.

The main reason of a protocol failure is hinted at by the structural classification of its attacks, especially their classification in the origin taxonomy:

Internal attacks imply that messages sent can be reused for multiple purposes. Such attacks can be repaired by introducing some manner of asymmetry in the protocol.

External attacks point at different problems, depending on their further classification.

- **Replay attacks** indicate that no freshness proof is included in the protocol. Such a freshness proof can for example be provided by the challenge-response pattern described before.
- **Interleaved attacks** when the challenge-response pattern is not instantiated correctly, it might allow an intruder to start up an interleaved attack. In such an attack, there is either confusion about the sender of a message, or about its receiver.

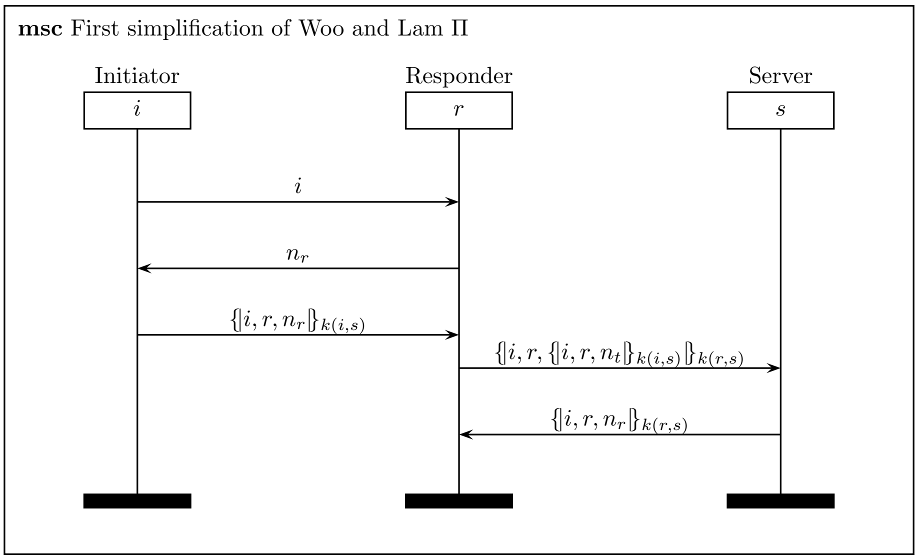

msc First simplification of Woo and Lam Π
Initiator
Responder
Server
i
r
s
i
n_r
$\{\!|i, r, n_r|\!\}_{k(i,s)}$
$\{\!|i, r, \{\!|i, r, n_t|\!\}_{k(i,s)}|\!\}_{k(r,s)}$
$\{\!|i, r, n_r|\!\}_{k(r,s)}$

Figure 5.11: First simplification of the Woo and Lam Π protocol

The sender and destination taxonomies give hints about what information is missing in a protocol, and is mainly useful for interleaved attacks.

If such an attack is also classified as a deflection attack, it means there is confusion about the receiver of a message, or that the intruder uses her dishonest agent, *Eve* to receive a message which is re-encrypted.

A fake sender-attack shows that the sender of a message is not fortified by the message itself, and that it can be reused in different runs.

A special class in the sender-destination taxonomy is the class of reflection attacks. These attacks show that it is clear who are the sender and receiver of the message, but that the roles of sender and receiver can be interchanged for the reflected message. We shall show this reflection attack in the example at the end this section.

Finally, the modification taxonomy shows where to repair a protocol. Those graphs which are classified as full-intercept have not been modified by the attacker in any substantial way. Especially for encrypted terms that are forwarded directly to the honest agents, a protocol can be mended by adding the sender or receiver identity. The missing identity follows from the classification in the sender and destination taxonomies, but can also be found by inspecting if an identity is already part of the message to be modified.

Woo and Lam Π

The Woo and Lam Π protocol [22] is simplified three times by its authors, and each of the three simplifications share an equivalent attack. We will focus on the first simplification of the protocol here, but mention the modifications that are introduced in the second and third simplification.

The first simplification of the protocol is given in Figure 5.11.

In the protocol, the initiator i authenticates herself to the responder r. To this end, a server is used that shares a key both with i and one with r. After i

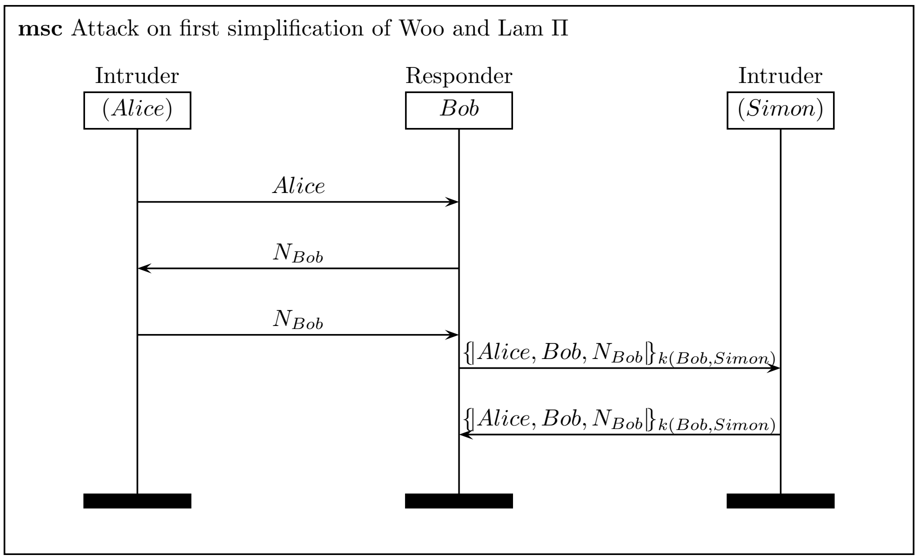

msc Attack on first simplification of Woo and Lam Π
Intruder
(*Alice*)
Responder
Bob
Intruder
(*Simon*)
Alice
N_{Bob}
N_{Bob}
$\{\|Alice, Bob, N_{Bob}\|\}_{k(Bob,Simon)}$
$\{\|Alice, Bob, N_{Bob}\|\}_{k(Bob,Simon)}$

Figure 5.12: Attack on the first simplification of the Woo and Lam Π protocol

starts the protocol by sending her name, the responder requests her to encrypt the newly generated nonce n_r. The initiator does this, generating a message that only the server s can decrypt. The message is subsequently encrypted by r, and sent on to the server. The server decrypts the messages, and re-encrypts the message generated by i. This translation is returned, and the responder decides that i was participating in the protocol.

Further simplifications of the protocol are based on the observation that the identity of r is not necessary in the messages. The only reason to include this would be for the benefit of the server, who already knows to be talking to r. Furthermore, the third message of the protocol ($\{\|i, n_i\|\}_{k(i,r)}$) does not need the identity of i, since r already knows he is communicating with i, and the server cannot derive it from the encrypted message.

All these versions of the protocol share the attack depicted in Figure 5.12.

This attack is an internal attack, as there is only one run (that of Bob), and it is also a *reflection* attack. All messages sent by *Bob* are reflected back to him, causing him to believe he communicated both with *Alice* and with *Simon*. The attack works because of two reasons. The first reason is that *Bob* cannot distinguish a message encrypted under a key not in his possession from any other random data. Since a nonce is also random data, he can accept his nonce N_{Bob} as the ticket from *Alice*. The second reason is that the message $\{\|Alice, Bob, N_{Bob}\|\}_{k(Bob,Simon)}$ that *Bob* subsequently sends to the server is exactly the message he expects back.

Based on the observation that the attack is a reflection attack, and that both the message N_{Bob} and the message $\{\|Alice, Bob, N_{Bob}\|\}_{k(Bob,Simon)}$ are reflected, we decide that we need some asymmetry in one or both of these messages. Since the first reflection is based on the fact that *Bob* sends random data and receives random data, we cannot really find a reparation in this message: to prevent the intruder modifying this at will, this would require some encryption by the

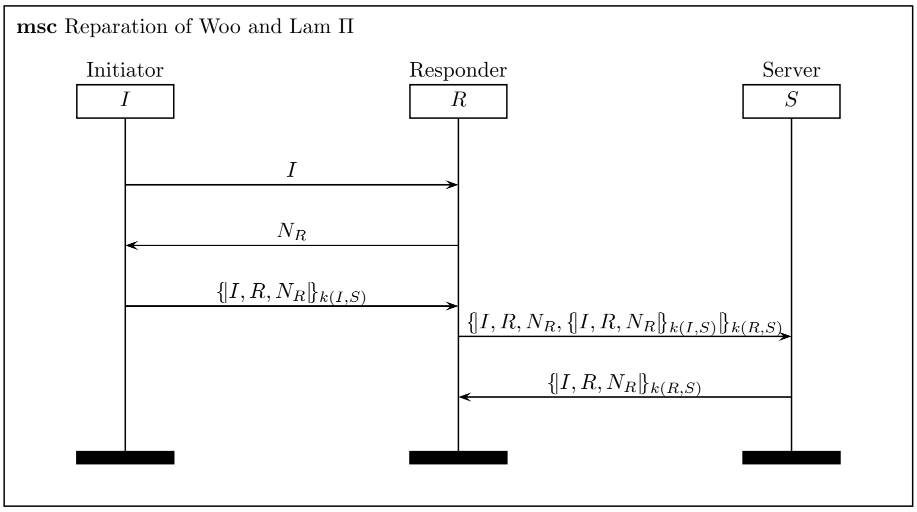

msc Reparation of Woo and Lam Π
Initiator
Responder
Server
I
R
S
I
N_R
$\{\!|I, R, N_R|\!\}_{k(I,S)}$
$\{\!|I, R, N_R, \{\!|I, R, N_R|\!\}_{k(I,S)}|\!\}_{k(R,S)}$
$\{\!|I, R, N_R|\!\}_{k(R,S)}$

Figure 5.13: The Woo and Lam Π protocol, repaired

responder, but this would then require more communication with the server. The second message is a more likely candidate for reparation, because there already is an encrypted message going back and forth.

By adding, for example, the responder's nonce N_R to the message, we would get the protocol in Figure 5.13. In this protocol, the message sent by R to S is no longer symmetrical to the message received by R from S. This frustrates the possibility of the intruder to reflect the message, and repairs the protocol. In fact, this protocol is the original Woo and Lam Π protocol, and was proved correct [22].

Chapter 6

Implementation

We have implemented the framework described in the previous chapters in the Python programming language [9]. The source code of this implementation can be found in Appendix A. We will give an overview of the program in this chapter.

6.1 Finding attacks: verifying protocols

The program first uses the Scyther backend to find as much attacks as possible. This is done by verifying the authentication goals of the protocols.

The program takes a directory filled with protocol descriptions in Scyther's security protocol description language (spdl files). If Scyther finds attacks against the protocol, these are returned as threads violating the property.

Scyther does not have a separate definition for an instantiated trace, using the normal trace for threads as well. Nonetheless, we refer to these instantiated objects as threads throughout this chapter.

The threads are grouped into runs, which are then analyzed further. Runs are created by checking for each thread if it matches the threads of any run already created. This matching means that the roles are played by the same agents, from the thread's point of view (so the role instantiation matches), and that for each message sent in one thread, and received in another thread, the instantiation is equal.

If the threads mismatch at any point, the evaluated thread is used to create a new run.

Scyther also represents the attacker's actions between runs as threads. We disregard these actions in the creation of runs, only allowing honest threads to constitute a run.

After runs have been created, they are analyzed structurally.

6.2 Analyzing attacks: structural

Classifying an attack according to the structural taxonomy is fairly straightforward. The difficult part in this is determining the interleaving of messages. This is done by first establishing the modifications.

The number of runs can easily be determined, since the runs are handed over to the analysis. This means that checking if an attack is an internal, is as easy as determining if the created set of runs is of size one.

If the attack is an external attack, the easiest way to check the interaction of the runs is by checking whether the runs are interleaved. If they are, the attack is an interleaved attack, while if they are not, the attack is a replay attack.

However, determining the interleaving of runs requires the modification to be established first. After these graphs have been completed, the program checks the interleaving by computing $\rightsquigarrow$. This is done by checking for each graph G starting in an honest read of run r_1, if its final state is an honest send in run r_2. This connection is also checked for the sends of run r_2 and the reads in run r_1. If this checks out for one send in r_1 and one send in r_2, then the runs are interleaved.

When constructing the runs, the program records the instantiation of roles inside it. This allows it to show the difference in instantiations between runs. The actual interpretation of the differences is left to the user, but could be further automated by grouping the send events and corresponding read events based on the (in)equality of sender and receiver.

For example, the NSPK protocol has two roles, i and r. Lowe's attack has two runs. The first run instantiates i to *Alice* and r to *Eve*, and the second run instantiates i to *Alice* and r to *Bob*. This means that any sends by *Alice* are deflected from *Eve* to *Bob*, while any sends from *Bob* in the second run are replayed straightly to *Alice*. This means that we can group the sends of *Alice* into the class of *deflection* attacks with respect to the destination taxonomy, and into the class of *straight forward* attacks with respect to the sender taxonomy. On the other hand, the send by *Bob* is grouped in the *straight replay* and *fake sender* classes.

6.2.1 Analyzing attacks: modifications

Constructing the modification graph is made easy because Scyther represents the actions of the intruder as short threads of one or more read events leading up to one send event.

This means that to create the graph, we need to connect each read event with a send event matching its message, and labelling the connection. If the send event was part of a dishonest thread, we need to match up the read(s) preceding this sent to the sends of either another dishonest thread, or in an honest thread. If a honest read can be matched to multiple sends, we give preference to a send of the intruder, as this could indicate the intruder forwarding some message.

Finally, the graph is completed once all its branches either hit an honest send or the intruder knowledge, which get translated to the final nodes of the graph.

The implementation of modified messages is a direct translation of the definitions of Chapter 4, without any difficult changes for the sake of implementation.

6.3 Results of implementation

The Scyther tool is distributed together with a formalization of almost all of the protocols found in SPORE. A few protocols are not formalized, because they use features not modelled in Scyther, such as the algebraic properties of exponentiation used in the Diffie-Helman protocol.

All other protocols of SPORE are formalized in Scyther's language, and we have tested our implementation against these protocols. This showed that most attacks on the protocols are external, interleaved attacks. Some attacks, however, were classified as internal attacks. Several of these attacks can be discarded as not being a very strong attack, but a result from Scyther's analysis method. Only the two attacks described in Chapter 5, the attacks on the TMN protocol and the Woo and Lam Π protocol, are interesting internal attacks.

Several of the disregarded internal attacks reported by Scyther are against an authentication property called *synchronization*, which we did not consider in our theory. This very strong property requires that all messages of a protocol are exchanged in lockstep, so that when a message is received by an agent at one moment, it has been sent strictly before this moment. Since our theory does not enforce this global order, the messages are grouped into one run.

For example, Lowe's modification of Denning and Sacco's shared key protocol [13], depicted in Figure 6.1, is vulnerable to an attack on synchronization. This attack abuses the fact that the first message of the protocol, the request from the initiator to the server, does not contain any timing information. This message could therefore be sent to the server long before the initiator starts the protocol. Due to the keys involved in the rest of the protocol, the intruder cannot break anything else.

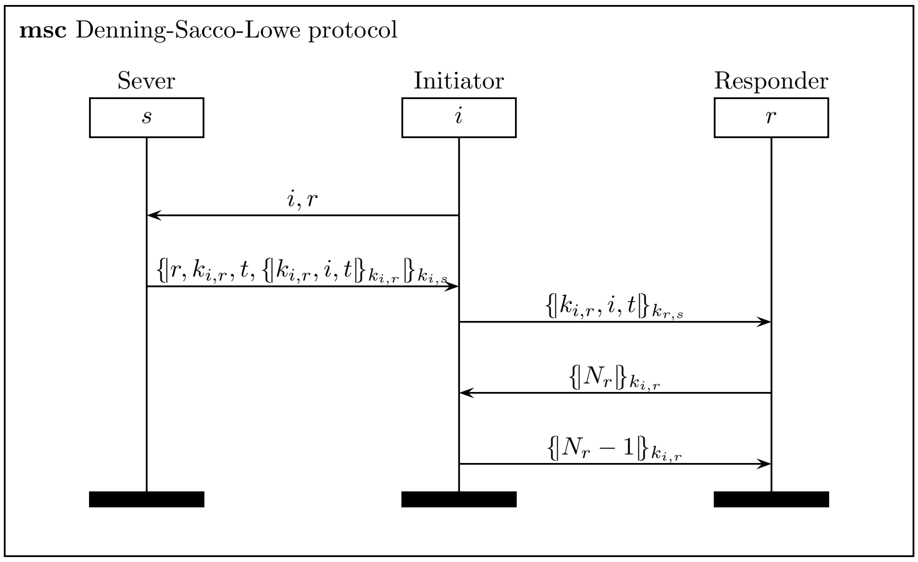

msc Denning-Sacco-Lowe protocol
Sever
s
Initiator
i
Responder
r
i, r
$\{\!|r, k_{i,r}, t, \{\!|k_{i,r}, i, t|\!\}_{k_{i,r}}|\!\}_{k_{i,s}}$
$\{\!|k_{i,r}, i, t|\!\}_{k_{r,s}}$
$\{\!|N_r|\!\}_{k_{i,r}}$
$\{\!|N_r - 1|\!\}_{k_{i,r}}$

Figure 6.1: Lowe's modification of Denning and Sacco's shared key protocol

Since the threads of server and initiator match after the protocol is completed, our program groups these threads into one run.

While it could be possible that an attack against synchronization is actually dangerous, the grouping into a single run which is a complete instantiation of the protocol, possibly missing the final read, seems to indicate that the intruder cannot do much harm. If she could do something, it would show as a difference in instantiation of one or more variables, or in the fact that more runs are necessary. The difference in instantiation, however, would already indicate an attack against agreement, instead of only synchronization.

In this chapter, we have shown our decisions in implementing a prototype tool based on our framework. We have used this tool to analyze attacks on the protocols in the SPORE library, obtaining the data necessary for Chapter 5. Next to this, we have shown that some *internal* attacks against synchronization might be disregarded safely.

Chapter 7

Conclusions

We have constructed a formal model for classifying attacks on security protocols. This taxonomy captures both the structural and modificational aspects of attacks. Using the model, it is possible to find similarities between attacks on different protocols. and in several cases recommend a protocol reparation based on the attack's class.

The research into this model can be divided into three aspects. The first is the development of the formal model itself, the second is the implementation of the model's framework in the Python programming language, and the third is the application of the model in reasoning about protocols and attacks.

7.1 Formal model

The formal model for attack classification is based on the concept of protocol runs. Each run is a consistent instantiation of a protocol, in which one or more honest agents participate. This implies that the honest agents participating in a run are actually communicating with each other, without interference of the intruder.

In our model, an attack is a combination of runs, between which a connection is made by the intruder. This connection consists of messages received in one run, which were not sent by an honest agent, but were constructed, or *modified* from messages sent in other runs. The classification of attacks then consists of determining how the runs are composed structurally, as well as determining what actions the intruder takes to modify messages sent.

For a structural classification, we have formalized the taxonomy formulated by Syverson [19], which is separated in an *origin taxonomy* and a *destination taxonomy.* The formalization mainly consisted of finding the notion of runs, which we formulated in such a way that we can define a set of predicates which determine where in the two taxonomies a given attack can be placed.

In giving this formalization, we noticed that the origination taxonomy did not describe how the sender of a message can be modified, but rather in what run a message was obtained (in the attacked run, in a parallel run or in a previous run). Therefore, we added a third dimension to the taxonomies, the *sender taxonomy.*

The structural taxonomy does not capture the actions of the attacker to modify messages, and especially what messages contribute in what way to the attack. To capture this, we introduce the notion of *modification graphs*, which builds, given an attack and a message, the way this message was constructed. Such a graph has two types of end nodes: one represents the initial attacker knowledge, the other type represents messages sent by honest agents.

A modification graph can then be classified according to a number of properties. A first method is determining if the modification graph for one message is *bisimilar* to the graph of another, either within the same attack, or for another attack. This could lead to a reduction in the number of modifications to analyze. A second classification of a graph tries to classify how messages from each run contribute to the construction of the message received. A class of modifications that is especially interesting, is the forwarding of a message directly into the receiving run. This can represent that the intruder could not construct the forwarded message, and had to extract it from another run. It is within this message that a protocol repair can be placed.

7.2 Implementation & application of model

To test and extend our framework, we implemented a prototype in Python. The experiments carried out with this implementation showed how the framework can be used in analyzing protocol flaws, based on the attacks on it, while the program itself shows that the framework lends itself well for implementation within a more sophisticated tool.

7.3 Open issues and future work

Even though we have shown that our framework is useful as is, there is still room for improvement. Specifically, the framework presented in this thesis can be seen as a foundation for a more sophisticated model for attack classification and analysis of this classification. To develop such a model, we suggest the following extensions, in order of importance:

1. Our framework can be used for generalizing attacks and finding a greatest common attack against a certain protocol goal. We have shown an example of such reasoning in the analysis of the TMN protocol in Section 6.3. However, we have not formalized the notions underlying this analysis. Towards this analysis, more research into equivalence of runs from different attacks is necessary, as well as analysis of semantical properties of modifications.

2. Currently, the experiment data is not post-processed, requiring a significant amount of manual analysis of the protocols. This seems a good first step for future research, since we believe that a fully automatic analysis of experiment data, including a grouping of attacks according to the classification, allows one to see (intuitive) equivalences between protocols more clearly. This post-processing should also give a more structured output.

3. As we already mentioned, we did not implement our model in the ProVerif tool, as this required a better knowledge of the inner working of the tool. This especially means we could not verify the usefulness of our model

for classifying replay attacks. To apply the theory to ProVerif, it is first necessary to explicitly extract the threads by honest agents, as well as the attacker actions from the attacks given by ProVerif. These threads can be subsequently grouped into runs, as we did in Scyther, for further analysis.

The fact ProVerif uses a process-algebraic representation of both honest and dishonest agents could be exploited to implement, and possibly further refine, our model and taxonomy of modifications.

4. Our implementation is not the most efficient. This efficiency is not a very large bottleneck, as analysis of the entire SPORE library, including the finding of attacks takes about one minute, and a full analysis of a single protocols, including finding the attacks, takes less than a second. However, if the model is implemented in a larger tool, it might become necessary to improve the efficiency

7.4 Reflection

Our approach of forming a model based on results from the SPORE library proved to be a useful one, especially for finding and formalizing a structural classification of the attacks. This also showed that the taxonomy of Syverson is a good one for determining the protocol flaws, despite missing one dimension. The formalization of this taxonomy did require finding a good notion of what a run is, something that was glossed over in the informal taxonomy.

Finding a suitable model for modifications was a more difficult task, however, as these modifications could fall into a wider range. Nonetheless, the graph approach looks promising in finding a reparation for a protocol, as it shows what messages were not modified by the attacker. This could give an indication of where a reparation would be most effective.

On a more global level, we might have obtained better results by doing analyses similar to those of Chapter 5 sooner and to more attacks, and extend our model and program to support these analyses.

Instead, we opted to first try and capture the intuition behind modification, and this left us with less time to analyze a larger set of attacks.

Furthermore, the output could have been better structured, as this would support a more rigorous analysis of the attacks. A structured output, however, would also require a more structured way of reading it, which could lead to another program for interpreting the output. As the tool's output was changing during the development, and we focused on validating the model first, we decided not to fix the output format yet.

Appendix A

Implementation source code

```
#!/usr/bin/python

"""
Experiment program to test structural/modification theory.
Author: Carst Tankink

"""

from Scyther import Scyther
from Scyther import Trace
from Scyther import Term
import sys
import os
import copy

# Class Definitions

class Transition(object):
  """
  A labeled transition between two states
  In our model, this represents the action taken by the intruder to obtain
  fromState by using toState
  """
  def __init__(self):
    self.fromState = None
    self.toState = None
    self.label = ""

class State(object):
  """
  A state represents an event in the construction of honest reads. We do not
  really care about whether it is a read or a send, with the exception of
  the initial state always being a honest read, but more about the term
  inside the event.
  """
  def __init__(self):
    self.event = None
    self.initial = False
    self.knowledge = False
    self.honest = False
    self.transitions = []
    self.id = 0

  def addTransition(self, trans):
    """
    Add the given outgoing trasition to the state
    """

    self.transitions.append(trans)


class Run(object):
  """
  Represents an honest, consistent run of the protocol
  """

  def __init__(self):
    self.roleInst = {}   # Role instantiation, roleInst: Role -> Agent
    self.events = {}   # Message instantiation, events: N ~> Event

  def isConsistent(self, thread):
    """
    Test if the given thread is consistent with this run
    """

    result = True

    # First test if role instantiation is consistent
    for key in self.roleInst.keys():
      if key in thread.roleAgents.keys():
        if self.roleInst[key] != thread.roleAgents[key]:
          result = False

    # Test
    for event in thread:
      strLabel = str(event.label)
      if strLabel in self.events.keys():
        if event.message != self.events[strLabel].message:
          result = False

    return result

  def addThread(self, thread):
    """
    Add the given thread to the run.
    """
    # pre: self.isConsistent(thread)
    # As the thread is consistent, we can just add the key-value pairs
    # of thread.roleAgents
    for key in thread.roleAgents.keys():
      self.roleInst[key] = thread.roleAgents[key]

    for event in thread:
      if isinstance(event, Trace.EventSend) or\
        isinstance(event, Trace.EventRead):
        self.events[str(event.label)] = event

# =============== end of class ===============

def paths(state):
  """
  Find all paths starting in state, and return them as a list of lists
  """

  if len(state.transitions) == 0:
    # The only path of a final state is the state itself
    return [ [state] ]
  else:
    result = []
    for tr in state.transitions:
      toPaths = paths(tr.toState)

      for p in toPaths:
        result.append([state] + p)

    return result


def classify(modification):
  """
  Classify the graph starting in state according to our taxonomy
  """

  # We need the paths in the graph for further analysis.
  p = paths(modification)

  # Find all final states of the graph. These final states are the last
  # states in all paths.
  # Also extract all states of the graph, again from the paths.
  finalStates =[]
  for path in p:
    final = path[len(path) - 1]

    if not final in finalStates:
      finalStates.append(final)



  # Test for constructive modification: determine if the (composed) message
  # was constructed, or forwarded directly.


  mforward = True
  # A modification is a forwarding modification if all paths to honest
  # states only branch using tupling.
  for path in p:
    if path[len(path) - 1].honest:
      for state in path:
        if len(state.transitions) > 1:
          for tr in state.transitions:
            if tr.label != "Tuple" and tr.label != "Untuple":
              mforward = False

  if mforward:
    print "The modification for %s" \
          "is a forwarding modification"%modification.event

  else:
    print "The modification for %s"\
          " is a constructive modification"%modification.event


  intercept = True
  fabrication = True
  # Test for full intercept: if the state leads only to honest sends
  for state in finalStates:
    intercept &= state.honest
    fabrication &= state.knowledge
    if intercept:
      print "Full_intercept"
    elif fabrication:
      print "Full_fabrication"
    else:
      print "Partial_intercept"

    # Determine if the modification has a single point of origin
    if not fabrication:
      honestFinals = [s for s in finalStates if s.honest]
      if len(honestFinals) == 1:
        print "Single_point_of_origin"
      else:
        print "Multiple_points_of_origin"

    # (if it is an intercept), or if it is a combination attack


  def connected(run1, run2, graphs):
    """
    Determine if an event in run2 is the final state of a graph starting in an
    event of run1
    """

    connected = False

    for (index,event1) in run1.events.items():
      if isinstance(event1, Trace.EventRead):
        graph1 = None

        # Find the graph starting in this event
        for graph in graphs:
          if graph.event == event1:
            graph1 = graph

        if not graph1:
          # Should not occur
          print "No_matching_modification_found."
          connected = False
        else:
          # Find all final events of graph1, through a DFS
          stack = [graph1]
          finalMessages = []

          while len(stack) > 0:
            toProcess = stack.pop()

            if not (toProcess.honest or toProcess.knowledge):
              stack += [transition.toState for\
                transition in toProcess.transitions]
            elif toProcess.honest:
              finalMessages.append(toProcess.event.message)

          # See if any event in run2 is in finalEvents
          for (index,event2) in run2.events.items():
            if event2.message in finalMessages:
              connected = True

    return connected

  def interleaved(runs, dishonest, filename, modifications):
    """
    Determines if (any) of the runs are interleaved, possibly using
    the dishonest actions
    """

    interleaved = False
    for r1 in runs:
      for r2 in runs:
        if r1 != r2:
          r1Tor2 = connected(r1, r2, modifications)
          r2Tor1 = connected(r2, r1, modifications)
          if r1Tor2 and r2Tor1:
            interleaved = True
    return interleaved

  def extendedSend(readState, honestRuns, dishonestRuns):
    """
    Subprocedure of determineModification, attempts to find a sendEvent in
    either honestRuns or dishonestRuns s.t. readState.message is a part of the
    found event's message. Return this event as a State
    """

    foundEvent = None
    honest = False

    for run in honestRuns:
      for (index, event) in run.events.items():
        if isinstance(event, Trace.EventSend):
          runTerms = []
          stack = [event.message]

          while len(stack) > 0:
            toProcess = stack.pop()

            if isinstance(toProcess, Term.TermTuple):
              stack.append(toProcess.op1)

              stack.append(toProcess.op2)
```

```python
          elif toProcess == readState.event.message:
            foundEvent = event
            honest = True

  for run in dishonestRuns:
    for event in run:
      if isinstance(event, Trace.EventSend):

        runTerms = []
        stack = [event.message]

        while len(stack) > 0:
          toProcess = stack.pop()

          if isinstance(toProcess, Term.TermTuple):
            op1 = toProcess.op1
            op2 = toProcess.op2
            stack.extend([op1, op2])

          elif toProcess == readState.event.message:
            foundEvent = event


  state = None
  if foundEvent:
    # Create a state out of the found event
    state = State()
    state.event = foundEvent
    state.honest = honest


  return state

def determineModification(runs, dishonest, filename):
  """
  Builds a modification graph for all read terms.
  """

  # Attempt to build a "knowledge graph" (transition system) for each
  # event in the honest threads
  graphs = []
  rid = 1

  for run in runs:
    for (key, read) in run.events.items():
      # Build a modification graph for each honest read
      if isinstance(read, Trace.EventRead):
        si = 1 # State index

        # The name of the graph file is the claim identifier plus the event
        # index.
        gn = filename + str(rid) + "." + str(read.index)
        rid += 1

        initialState = State()
        initialState.event = read
        initialState.initial = True
        initialState.id = 0

        # Commence a DFS to find events leading up to this event.
        toProcess = [initialState]

        # Keep a second record of all states discoverd, to make connections
        states = [initialState]

        while len(toProcess) > 0:

          readState = toProcess.pop()

          sendState = None
          read = readState.event


          dishonestThread = None
          honestRun = None
          sendEvent = None

          # First, search for corresponding send in dishonest thread
          for thread in dishonest:
            for event in thread:
              if isinstance(event, Trace.EventSend) and\
                 event.message == read.message:
                dishonestThread = thread


                sendEvent = event

                # Check if the matching state is already part of the graph
                for state in states:
                  if sendEvent == state.event:
                    sendState = state

                # If not, construct a new state.
                if not sendState:
                  sendState = State()
                  sendState.event = sendEvent
                  sendState.id = si
                  si += 1
        states.append(sendState)

# Add the events of the dishonest thread to the graph.
if dishonestThread and sendEvent:
  for event in dishonestThread:
    if event.index < sendEvent.index:
      if isinstance(event, Trace.EventRead):
        transition = Transition()

        readState1 = None
        for state in states:
          if state.event == event:
            readState1 = state
        if not readState1:
          readState1 = State()
          readState1.event = event
          readState1.id = si
          si += 1

          toProcess.append(readState1)
          states.append(readState1)

        # Attach states to graph
        transition.fromState = sendState
        transition.toState = readState1

        # Determine label for the transitions
        if isinstance(sendEvent.message, Term.TermEncrypt):
          if sendEvent.message.value == event.message:
            transition.label = "plain"
          elif sendEvent.message.key == event.message:
            transition.label = "key"
          else:
            transition.label = "?"
        elif isinstance(sendEvent.message, Term.TermApply):
          if sendEvent.message.function == event.message:
            transition.label = "function"
          elif sendEvent.message.argument == event.message:
            transition.label = "argument"
          else:
            # This case should not occur
            transition.label = "?"
        else:
          # In this case,
          transition.label = "??"

        sendState.addTransition(transition)



  for tr in sendState.transitions:
    if tr.label == "??":
      if isinstance(tr.toState.event.message, Term.TermEncrypt):
        tr.label = "Decrypt-Term"
      else:
        tr.label = "Decrypt-Key"


else:
  # No matching dishones actions were found, so try to find a
  # matching send in the honest threads.
  for run1 in runs:
    for (key, send) in run1.events.items():
      if isinstance(send, Trace.EventSend):
        if send.message == read.message:
          sendEvent = send
          honestRun = run1

          for state in states:
            if state.event == sendEvent:
              sendState = state

          if not sendState:
            sendState = State()
            sendState.event = sendEvent
            sendState.honest = True
            sendState.id = si
            si += 1
            states.append(sendState)

if not sendEvent:
  # Still no matching event found, so scan the initial knowledge
  # for occurence of the term.

  # The initial knowledge is sent as a single tuple. Break it down
  # into individual terms.
  initialSend = dishonest[0].eventList[0]
  sendTerms = []
  unprocessed = [initialSend.message]

  while len(unprocessed) > 0:
    msg = unprocessed.pop()
    if isinstance(msg, Term.TermTuple):
      unprocessed.extend([msg.op1, msg.op2])
    else:
      sendTerms.append(msg)

  # Check if the message is fully part of the initial knowledge
  if read.message in sendTerms:
    sendEvent = initialSend

    for state in states:
      if state.event == sendEvent:
        sendState = state


    if not sendState:
      sendState = State()
      sendState.event = sendEvent
      sendState.knowledge = True
      sendState.id = si
      si += 1

      states.append(sendState)

  # If all else fails, break down honest tuples into its
  # constructors.
  if not sendEvent and isinstance(read.message, Term.TermTuple):
    # Create event for op1 and op2, process

    read1 = copy.copy(read)
    read2 = copy.copy(read)
    read1.message = read.message.op1
    read2.message = read.message.op2

    readState1 = None
    readState2 = None
    for state in states:
      if state.event == read1:
        readState1 = state
      if state.event == read2:
        readState2 = state
    if not readState1:
      readState1 = State()
      readState1.event = read1
      readState1.id = si
      si += 1
      states.append(readState1)

    transition1 = Transition()
    transition1.fromState = readState
    transition1.toState = readState1

    transition1.label = "Tuple"
    readState.addTransition(transition1)

    if not readState2:
      readState2 = State()
      readState2.event = read2
      readState2.id = si
      si += 1
      states.append(readState2)

    transition2 = Transition()
    transition2.fromState = readState
    transition2.toState = readState2
    transition2.label = "Tuple"
    readState.addTransition(transition2)


    toProcess.extend([readState1, readState2])

# If the read term comes from an honest send, we have an intercept
# transition.
if readState and sendState:
    transition = Transition()
    transition.fromState = readState
    transition.toState = sendState

    if sendState.honest:
      transition.label = "Intercept"
    elif sendState.knowledge:
      transition.label = "Obtain"
    else:
      transition.label = "A"

    if readState.id != sendState.id:
      # Somehow, a selfloop is introduced somewhere. We don't want that.
      readState.addTransition(transition)

if readState and len(readState.transitions) == 0:
  # If there are no transitions found, the read event was part of
  # a larger tuple.
  sendState = extendedSend(readState, runs, dishonest)

  if sendState:
    foundState = False

    for state in states:
      if state.event == sendState.event:
        foundState = True

    if not foundState:
      sendState.id = si
      si += 1
```

```
                states.append(sendState)

                if not sendState.honest:
                  toProcess.append(sendState)


                # Build a untuple transition from readState to sendState
                transition = Transition()

                transition.fromState = readState
                transition.toState = sendState


                transition.label = "Untuple"

                readState.addTransition(transition)

        graphs.append(initialState)

    exportGraph(states, gn)

  return graphs

def exportGraph(states, gn):
  """
  Export the graph of states to an XML gn.xml file readable by LTSgraph
  """

  x = -1000
  y = -1000

  # Write graph to an LTSgraph xml file
  gn += ".xml"
  graphFile = open(gn, 'w')


  graphFile.write('<?xml␣version=\"1.0\"␣?>\n')
  graphFile.write('<Graph>\n')

  stateString = "<State␣value=\"%d\"␣isInitial=\"%s\"␣x=\"%d\"␣y=\"%d\""\
    "red=\"255\"␣green=\"255\"␣blue=\"255\">\n"
    "<Parameter␣name=\"event\">%s</Parameter>\n</State>\n"

  transitionString = "<Transition␣from=\"%d\"␣to=\"%d\""\
    "label=\"%s\"␣x␣=␣%d␣y␣=␣%d/>\n"
  while len(states) > 0:
    x += 150
    while x > 1000:
      x -= 1000
    y += 150

    while y > 1000:
      y -= 1000

    state = states.pop(0)

    if state.initial:
      initial = "1"
    else:
      initial = "0"

    graphFile.write(stateString%(state.id, initial, x, y, state.event))
    for transition in state.transitions:
      graphFile.write(transitionString%(state.id,\
       transition.toState.id, transition.label,x + 75, y + 75))

  graphFile.write("</Graph>\n")

def groupThreads(threads):
  """
  Group threads into runs.
  """

  result = [] # List of the runs created by grouping

  for thr in threads:
    createNewRun = True

    # Run identifier
    i = 0
    for run in result:
      if run.isConsistent(thr):
        createNewRun = False
        run.addThread(thr)
      i += 1

    # The examined thread was not consistent with any run yet, so it is the
    # basis for a new run.
    if createNewRun:
      newrun = Run()
      newrun.addThread(thr)
      result.append(newrun)

  return result

def analyze(attack):
  """
  Analyze structure of attack.
  """
  # First, find all threads of the attack.
  # Threads are called runs in Scyther, and returned as a part of the attack
  # this is just a renaming
  threads = attack.semiTrace.runs

  # Print preliminary information about the attack to stdout
  print "-␣Attack␣" + str(attack.id) + "␣has␣" + str(len(threads))  + "␣threads."

  # Split threads into honest and dishonest threads
  honestThreads = []
  dishonestThreads = []

  for thr in threads:
    if not thr.intruder:
      honestThreads.append(thr)
    else:
      dishonestThreads.append(thr)

  print "␣...␣of␣these,␣" + str(len(honestThreads)) + "␣are␣honest."

  protDescr = attack.protocoldescr

  # Group honest threads into runs
  runs = groupThreads(honestThreads)

  # Determine modifications of the attack
  modGraphs = determineModification(runs, dishonestThreads,\
    str(attack.claim) + str(attack.id)+ ".")


  if len(runs) == 1:
    print "␣*␣Internal␣attack"
  else:
    print "␣*␣External␣attack␣(%d␣runs)" % len(runs)
    if interleaved(runs, dishonestThreads, str(attack.claim), modGraphs):
      print "␣␣␣*␣Interleaved␣attack"
    else:
      print "␣␣␣*␣Replay␣attack"

  tested = {}

  for run in runs:
    tested[run] = False

  for run1 in runs:
    tested[run1] = True
    for run2 in runs:
      if not tested[run2]:
        for (role,agent) in run1.roleInst.items():
          if role in run2.roleInst.keys():
            print "Role:␣%s␣is␣instantiated␣in␣run␣1␣as␣%s,"\
              "␣and␣as␣%s␣in␣run␣2"%(role, run1.roleInst[role], run2.roleInst[role])
          else:
            print "Role␣%s␣is␣not␣instantiated␣in␣run␣2"%role
    if len(runs) == 1:
      for (role, agent) in run1.roleInst.items():
        print "Role␣%s␣is␣instantiated␣as␣%s"%(role, agent)

  # Report on modification classifications
  for state in modGraphs:
    #print "Modification for: %s"%state.event
    classify(state)

  return modGraphs

def findAttacks(s, protocol):
  """
  Given a protocol file, find all attacks on it.
  """
  print "Looking␣for␣attacks␣on␣" + protocol

  s.setFile(protocol)
  s.verifyOne()

  graphs = []
  for claim in s.claims:
      #if claim.claimtype != "Secret" and not claim.okay:
      if not claim.okay:
        #print "Broken claim: "+ str(claim)

        for attack in claim.attacks:
          # Analyze the attack in another function, as it is quite involved.
          graphs += analyze(attack)

  print "=========================================================================="

  return graphs

def main(pars):
    """
    Run Scyther analysis on the protocols in the given directory (pars[1]) and
    analyze the result based on the predicate given in pars[0].
    """

    directory = pars[0]
    attacks = []
    s = Scyther.Scyther()
    # For now, do typed matching
    s.options = "--match=0 -A --max-attacks=10 --max-runs=5"
    filelist = os.listdir(directory)

    for fname in filelist:
        if fname.find(".spdl") >= 0:
          protocolAttacks = findAttacks(s, directory + fname)
          if len(protocolAttacks) > 0:
            attacks += protocolAttacks


if __name__ == '__main__':
    pars = sys.argv[1:]
    main(pars)


# vim: set ts=2 sw=2 et list lcs=tab\:>-:
```

Bibliography

[1] Ross J. Anderson. *Security Engineering – A Guide to Building Dependable Distributed Systems*. Wiley, 1st edition, 2001. Available throgh the author's webpage: `http://www.cl.cam.ac.uk/~rja14/`. (1)

[2] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society. (16)

[3] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007. (17, 39)

[4] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990. (17, 40)

[5] Cas J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006. (4, 16)

[6] Cas J.F. Cremers. Unbounded Verification, Falsification and Characterization of Security Protocols by Pattern Refinement. *Proceedings of the 15th ACM conference on Computer and Communicatsions security*, page 10, August 2008. (8, 16)

[7] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992. (14)

[8] Danny Dolev and Andrew Chi-Chih Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983. (7, 13)

[9] Python Software Foundation. Python programming language – official website. Webpage, obtained through `http://www.python.org` on June 10, 2009. (51)

[10] Alan Jeffrey and Christian Haack. Cryptyc – Cryptographic Protocol Type Checker, 2004. Webpage, obtained through `http://cryptyc.cs.depaul.edu` on February 13, 2009. (17)

[11] David Kahn. *The CodeBreakers: The Comprehensive History of Secret Communication from Anicent Times to the Internet*. Scribner, 1996. (1)

[12] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996. (2)

[13] Gavin Lowe. A Family of Attacks upon Authentication Protocols. Technical Report 1997/5, University of Leicester, 1997. (53)

[14] Gavin Lowe. A Hierarchy of Authentication Specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 31–43. IEEE Computer Society Press, 1997. (15)

[15] Gavin Lowe and A. W. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23(10):659–669, 1997. (44)

[16] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM*, 21(12):993–999, 1978. (2, 11)

[17] Dawn Song, Sergey Berezin, and Adrian Perrig. Athena: a Novel Approach to Efficient Automatic Security Protocol Analysis. *Journal of Computer Security*, pages 191–230, 2001. (16)

[18] SPORE. SPORE – Security Protocols Open Repository. Webpage, obtained through `http://www.lsv.ens-cachan.fr/Software/spore/` on February 16, 2009. (4, 16)

[19] Paul Syverson. A Taxonomy of Replay Attacks. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 187–191. Society Press, 1994. (iii, 4, 21, 55)

[20] Makoto Tatebayashi, Natsume Matsuzaki, and David B. Newman Jr. Key distribution protocol for digital mobile communication systems. In *Advances in Cryptology — CRYPTO '89*, volume 435 of *LNCS*, pages 324–333. Springer-Verlag, 1989. (43)

[21] F.J. Thayer, J.C. Herzog, and Joshua D. Guttman. Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security*, 7(2/3), 1999. (16)

[22] Thomas Y. C. Woo and Simon S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 28(3):24 – 37, 1994. (47, 49)