EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Systematic analysis of attacks on security protocols

Hollestelle, G.

Award date:
2005

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER'S THESIS

Systematic Analysis of Attacks on Security Protocols

by
G. Hollestelle

Supervisors:

dr. S. Mauw
ir. C.J.F. Cremers

Eindhoven, November 2005

Abstract

This thesis describes a method to analyze attacks on security protocols. It provides a formal description of the method used and a description of the results that have been obtained by using an automated version of this method on protocols from literature in isolation and in a multi-protocol setting.

Preface

This document presents my master thesis for Computer Science And Engineering at the Eindhoven University of Technology. The research and work was done between April 2005 and November 2005 within the Formal Methods group of the Department of Mathematics and Computer Science. My supervisors were Dr. Sjouke Mauw and Ir. Cas Cremers. Other members of the committee were Prof. Jos Baeten and Dr. Mohammad Mousavi.

Acknowledgements

I am very grateful to Cas Cremers for his supervision during this graduation project and for giving me the freedom to take this project into the direction that I found most interesting. Without his assistance this thesis would never have been written. I would also like to thank Sjouke Mauw for providing me the opportunity to graduate at the ECSS group and for helping me to better structure my thesis. I would also like to thank Jos Baeten and Mohammad Mousavi for being so kind to join my committee on a relatively short notice.

Gijs Hollestelle

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Research Question . . . 2
1.3 Document Structure . . . 2

2 Security Protocols in Detail **4**
2.1 Informal Introduction to Security Protocols . . . 4
2.1.1 Basic Concepts . . . 5
2.1.2 Protocol Execution . . . 6
2.1.3 The Intruder . . . 7
2.1.4 Security Claims . . . 7
2.2 Attacks . . . 8
2.2.1 Attack Example: Man in the Middle . . . 8
2.2.2 Attack Example: Tickets and Type Flaws . . . 9
2.2.3 Attack Example: Key Compromise . . . 10

3 Systematic Analysis of Attacks: a Domain Analysis **12**
3.1 Agent Model . . . 13
3.1.1 Implementation Details . . . 13
3.1.2 Agent Behaviour . . . 14
3.1.3 Executed Protocols . . . 14
3.1.4 Executed Roles . . . 15
3.1.5 Runs . . . 15
3.1.6 Initiation . . . 16
3.2 Threat Model . . . 17
3.2.1 Network Capabilities . . . 17
3.2.2 Untrusted Agents . . . 17
3.2.3 Intruder Complexity . . . 17
3.3 Cryptographic Primitives . . . 18
3.4 Security Requirements . . . 18

4 Formal Definitions of Properties **19**
4.1 Notation . . . 19
4.2 Definitions . . . 20
4.2.1 General Definitions . . . 20
4.2.2 Definitions Related to Actions and Roles . . . 20
4.2.3 Definitions Related to Runs . . . 21
4.3 Consequences . . . 22

4.4 Requirements . . . 22
4.4.1 Type Flaws . . . 22
4.4.2 Agent Roles . . . 23
4.4.3 General Requirements . . . 23
4.4.4 Intruder Capabilities . . . 23
4.4.5 Feasibility . . . 23
4.5 Attack Complexity . . . 24

5 Experiments **25**
5.1 Experiment Goals . . . 25
5.2 The Experiment Tool . . . 25
5.2.1 Analysis Phase . . . 26
5.2.2 Classification Phase . . . 26
5.2.3 Filtering and Sorting Phase . . . 26
5.3 Input Sets . . . 26
5.3.1 SPORE . . . 27
5.3.2 Multi Protocol . . . 27
5.4 The Scyther Tool . . . 28
5.4.1 History and Development . . . 28
5.4.2 Technical Details . . . 28
5.4.3 Attack Representation . . . 28
5.4.4 Current Limitations of The Scyther Tool . . . 30
5.4.5 Working Around Current Limitations . . . 30
5.5 Experiment Execution . . . 31
5.5.1 Performance . . . 31
5.6 General Results . . . 31
5.6.1 Identifying Identical Attacks . . . 31
5.6.2 Structuring Large Collections of Attacks . . . 32
5.6.3 Filtering Attacks . . . 32
5.7 Results Specific to SPORE . . . 32
5.7.1 Without Type Flaws . . . 32
5.7.2 With Full Type Flaws . . . 32
5.8 Results Specific to Multi Protocol . . . 33
5.8.1 Without Type Flaws . . . 33
5.8.2 With Full Type Flaws . . . 33

6 Miscellaneous Results **34**
6.1 Passive Key Compromise . . . 34
6.2 Newly discovered Attacks . . . 35
6.2.1 Single Protocol . . . 35
6.2.2 Multi Protocol . . . 37

7 Conclusion **39**
7.1 Future Research . . . 39
7.2 Practical Applicability . . . 40

Bibliography **41**

Chapter 1

Introduction

This chapter introduces the research question that this research tries to answer and it contains details about the structure of the document.

1.1 Background

In the modern information age, the value of information in our society is growing. Many processes that were once controlled by physical actions are now completely automated. An example of this is the transition by banks from physical storage of money and gold to being completely based on digital information. The increase in the value of information also increases the need for information security.

The wide spread adoptation of the Internet has introduced a large number of applications, fully using the possibilities of digital communications. The Internet can nowadays be used for all kinds of applications that involve the transfer of information, from downloading electronic music to on-line banking systems. In general, these applications use a prescribed protocol in order to make communication possible. Apart from the operational goals of such a protocol there are often also requirements on the security of the information that is being transfered. This research focusses on the security aspects of protocols. We consider protocols that have certain security goals and call these protocols security protocols.

In practice, it has proven to be very difficult to design correct security protocols and it can take a long time before it is discovered that a protocol that has been used for an extensive amount of time is in fact incorrect and does not satisfy the requirements that it was supposed to satisfy. At first, security protocols were analyzed in an informal way, later this analysis was formalized, for example by Burrows, Abadi and Needham in [BAN96]. Nowadays formalized methods are combined with computer tools that can analyze the security of protocols automatically. In this setting a protocol can be proven to be correct or a series of attacks can by found, that prove its incorrectness. Attacks are valid executions of the protocol (i.e. following the rules of the protocol) but not satisfying the security goals that should be guaranteed by the protocol.

1.2 Research Question

The research of security protocols has yielded a large collection of attacks on protocols. Even now there are often claims of newly discovered attacks, that have not been reported before. This brings us to the questions that this research tries to answer. What doest it mean for an attack to be different from another attack? When a new attack is discovered, how can we be sure that it is actually a new attack? Why are attacks considered to be attacks? Does it break new security assumptions, does it break them in a different way? How do we know that an attack is relevant in a certain environment?

These questions ask for a systematic way in which attacks can be analyzed. Often a new attack is published simply by giving the protocol execution that leads to the broken security assumptions. Seldom does the description of the attack explicitly mention what assumptions are needed for the attack to work, e.g. the environment in which the protocol operates or the intruder model that has been used. A trained observer might be able to manually deduce these assumptions from the attack, but it would be favorable to be able to do so in a more structured or automated way.

Apart from the requirements that are often unclear from an attack the consequences of an attack may not be made explicit by the person that has constructed the attack either. In literature the consequences of attacks can vary from being able to obtain large amounts of encrypted text to the intruder being completely capable of impersonating honest agents and obtaining information that should have been kept secret. Because of this wide variety of consequences, it is important that the exact consequences of an attack can be derived from it.

All of this leads to the following research question that this research tries to answer:

Given an attack, how can we find the essence of the attack?

By essence we mean what makes an attack different from another attack and what makes an attack interesting. In order to answer this question we will use a systematic approach to identify attack related properties that capture the essence of attacks. The resulting classification of attacks based on these properties must be useable for three purposes:

- Be able to identify if a given attack is equal to another given attack
- Be able to structure the huge amount of output generated by automated security protocol checkers by defining an order on attacks and eliminating duplicate attacks
- Be able to filter out attacks that are not relevant in a certain environment

1.3 Document Structure

In order to answer the research question mentioned in the previous section, the basis of security protocols and attacks need to be clear. The environment in which the research question is answered, is discussed in Chapter 2 clarifying

the underlying model and the basis of security protocols. Chapter 3 and Chapter 4 describe a systematic method to analyze attacks based on certain properties, related to the concepts that have been explained in the preceding chapter. Chapter 5 describes the process of automating the classification method using a combination of existing and newly developed tools, and the application of this automated method on two input sets of protocols. Chapter 6 describes miscellaneous results obtained while performing the research. Chapter 7 contains the conclusions and recommendations for the directions of future research.

Chapter 2

Security Protocols in Detail

In order to give the reader a better understanding of the research question, this chapter describes the environment in which this question is answered. In this section a precise but informal textual description of the involved concepts is given. Formal definitions of these concepts are given by the semantics described in [CM05].

2.1 Informal Introduction to Security Protocols

Modern communication applications have certain security requirements with repsect to the communication channels, for example that messages can not be altered and are kept confidential from a possible attacker. A security protocol describes a method to obtain a channel that has these properties in a potentially hostile network environment such as the Internet.

The protocol below will be used to introduce the concepts related to security protocols; it will be explained in the next sections.

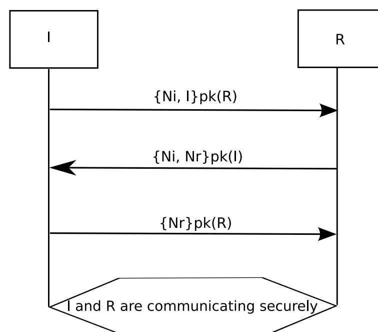

Figure 2.1: Informal Description of the Needham Schroeder Protocol

2.1.1 Basic Concepts

The following concepts are used to describe security protocols:

- Roles
  A protocol describes a series of actions for certain roles. Roles are denoted using a single capital letter, for example I (for initiator), R (for responder) and S (for server). In general, the initiator establishes a connection with a responder, in some cases via a server. The protocol described in Figure 2.1 consists of two role descriptions: I and R.

- Agents
  When a protocol is executed, the roles in the protocol description are performed by agents. An agent can perform any number of roles in a protocol execution. For illustrative purposes agents are often denoted using human names such as Alice, Bob and Simon.

- Knowledge
  An agent starts a protocol execution with a certain initial knowledge, this initial knowledge often consists of a number of keys that are shared with other agents. When an agent executes a certain role it is possible that the agent creates a number of values that are also added to its knowledge. An agent expands its knowledge during the protocol execution by receiving data from the network.

- Messages
  During the execution of the protocol agents exchange messages via the network. Agents can only send messages that are in their knowledge and when they receive a message, that message is added to their knowledge.

- Encryption
  In order to send a value in such a way that only the intended recipient can read it, the concept of encryption is used. From now on, the encryption of message m with key k will be denoted as $\{m\}_k$. Here k is called the encryption key, as it is used for encryption. The key that is required for decryption of a message that has been encrypted using k will be denoted as k^{-1} (or the inverse of key k). In symmetric key environments k is equal to k^{-1} where in public key environments k (the public key) is not equal to k^{-1} (the secret key). In this report we will assume that all encryption is done using a perfect encryption scheme, resulting in the following assumption:

  Assumption 1 *The message m can only be obtained from $\{m\}_k$ by agents that know k^{-1}.*

- Nonces
  In order to make sure that a message has been generated recently by an agent, so called nonces can be used. Nonces are randomly generated numbers that will only be used once, thereby making the messages exchanged in the protocol unique for every execution of the protocol. We will assume that no one is able to guess or predict nonce values.

2.1.2 Protocol Execution

When agents perform roles in a protocol, we get a protocol execution or trace. An example of a possible protocol execution for the Needham Schroeder protocol is given below:

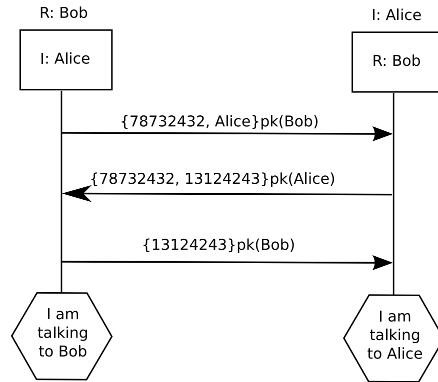

Figure 2.2: Example Execution of the Needham Schroeder Protocol

In Figure 2.2 agent names Alice and Bob have been instantiated for the role names, Alice performs role I and Bob performs role R. Ni and Nr have been instantiated by randomly generated nonce values, created specifically for this execution of the protocol.

A trace gives an overview of all the participants in the system. However the agents themselves can only observe their own status. Meaning that in Figure 2.2 Alice can only see her own run and knows nothing of Bob except for the messages she is apparently receiving from him. The view that an agent has on the execution of the protocol is called a run, the runs of Alice and Bob are depicted below.

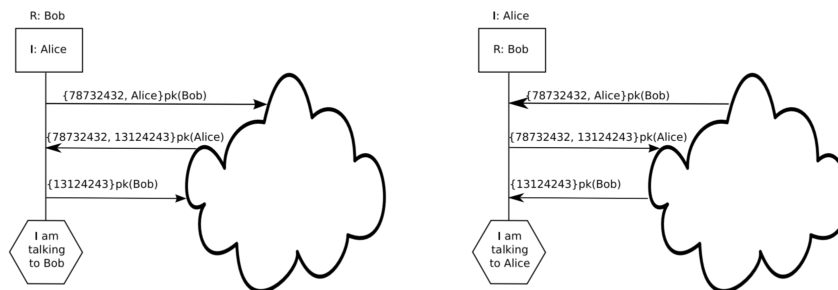

Figure 2.3: Two Example Runs of the Needham Schroeder Protocol

2.1.3 The Intruder

Security protocols are used to protect communications from an intruder, we distinguish three different kinds of intruders:

- Passive
  A passive intruder is able to eaves drop on all messages sent while executing the protocol.
- Active
  An active intruder has the same capabilities as a passive intruder has, but also has full control over the network. It means that the intruder is able to inject messages into the network, block messages and to alter messages. We call this the Dolev-Yao intruder model [DY83]. An active intruder is also able to compromise previously used session keys within a reasonable amount of time.
- Dishonest agents
  We also take into account the fact that there can be regular agents that work together with the intruder. This can happen for a variety of reasons, for example because the agents themselves are dishonest or because their computer has been compromised. From now on, we will call agents that are conspiring with the intruder untrusted agents.

2.1.4 Security Claims

In order to describe the security goals that a protocol tries to accomplish, local security claims will be used. When an agent reaches a point in the execution of the protocol where a security claim can be issued, he believes that there are no protocol traces that can reach the claim, for which the claim does not hold.

If one or more of the agents involved in the run is an untrusted agent, the claims of that run are not of interest any more, because they can always be violated by the untrusted agent. Therefore only claims in runs for which only trusted agents are involved are considered. The following security claims will be considered in this research:

Secrecy

One of the properties that is often desired of a security protocol is for it to keep certain values secret. A Secrecy claim of a certain value v in a trace is valid, if at the end of the trace the intruder does not know the value v.

Freshness

Security protocols often have a goal to accomplish a session key, apart from the fact that this key has to be kept secret, it is also required that this key has not been used before. For this purpose, we introduce the Freshness claim. A Freshness claim of a certain value v in a trace is valid, if there is no other Freshness claim of the same value v performed by the same role.

Authentication

Another important security requirement of a protocol is that a protocol should guarantee that all parties that should be active according to the protocol specification are active. And that roles have been performed by the agents that claim to have performed them. In order to reason about these properties, a number of so called authentication claims are introduced as follows. The description of the first claim originates from [Low97] the description of last two from [CMdV03].

- Liveness
  A Liveness claim of a certain agent a in a trace is valid, if that agent a has performed an action in the trace, before the claim.

- Strong Liveness
  In order to distinguish a bit more between agreement and liveness related claims we introduce a new stronger liveness claim that will be called Strong Liveness. A strong liveness claim of an agent a and role r in a trace is valid if agent a has performed an action in role r in the trace, before the claim.

- Non Injective Agreement
  A Non Injective Agreement claim in a trace is valid if all actions before the claim have been performed according to the protocol specification.

- Non Injective Synchronization
  A Non Injective Synchronization claim in a trace is valid if all actions before the claim have been performed according to the protocol specification and all send events have occurred before their corresponding read events.

Injectivity

In the descriptions of authentication claims above we only used the non injective version of agreement and synchronization. In the literature there are also injective variants of these claims [CMdV03]. The injective variants of these claims are stronger and also require that there is a unique one-to-one mapping of runs. In this research we do not address Injectivity and focus only on non-injective claims.

2.2 Attacks

All traces that contain at least one broken claim are considered to be attacks, in the next sections three examples of attacks will be given and some of the interesting concepts related to attacks are introduced.

2.2.1 Attack Example: Man in the Middle

The attack in Figure 2.4 shows the famous man in the middle attack on the Needham Schroeder protocol as discovered by Lowe in 1995 [Low95]. The attack uses the fact that Alice starts a session with the untrusted agent Eve to impersonate Alice in a new session with a third agent Bob. Alice has no way of noticing that this is happening as all she can observe is a regular run with

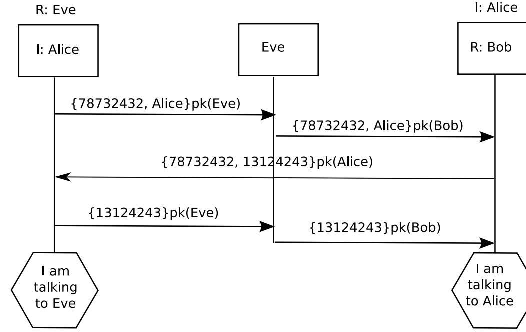

Figure 2.4: Man in the Middle Attack on Needham Schroeder

agent Eve. Also note that Bob's run is equal to the run depicted in Figure 2.3, meaning that for Bob the situation in which Alice is actually talking to him is indistinguishable from the one in the attack, where Eve is impersonating Alice.

This attack uses an active intruder and an untrusted agent (Eve) that conspires with the intruder. We also see that the intruder has to wait for Alice to start a session with the untrusted agent Eve, we will call this concept initiation. Also observe that only the responder (Bob) is being attacked, Alice's claim that she is talking to Eve is not considered to be broken. Note that this attack does not break (Strong) Liveness for any agent.

The central theme in this thesis will be the analysis of attacks based on properties as described above for the example attack.

2.2.2 Attack Example: Tickets and Type Flaws

In order to illustrate tickets and type flaws a new protocol is introduced: the WooLam-Pi1 protocol, as described by Woo and Lam in [WL94].

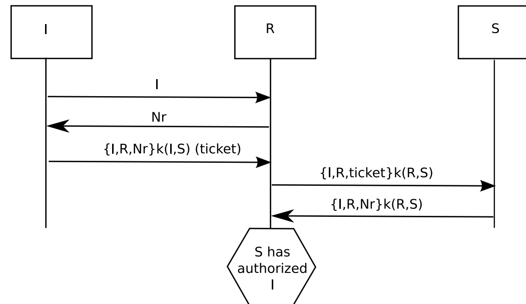

Figure 2.5: Informal Description of The WooLam-Pi1 Protocol

The WooLam-Pi1 protocol allows the responder to verify the identity of the initiator, via a server without the need of sharing a key with the initiator. In order to accomplish this a message that is intended for the server is generated by

the initiator and sent to the responder. The responder can not observe the contents of this message, because it is encrypted with a key that only initiator and server share. Messages that are received by an agent but can not be decrypted by that agent are called tickets. In this protocol the responder simply encrypts the ticket he receives, together with the names of initiator and responder and forwards it to the server.

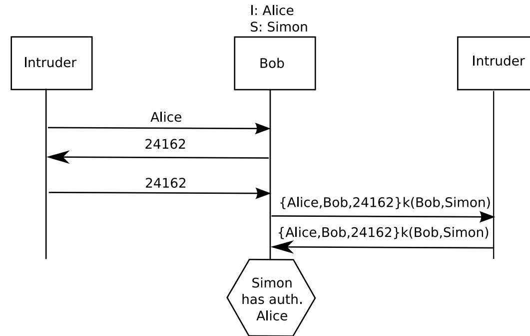

Figure 2.6: Attack on the WooLam-Pi1 Protocol

Figure 2.6 shows an attack on the WooLam-Pi1 protocol, that uses the assumption that in addition to being unable to observe the contents of the ticket, the responder is also unable to observe the type of the ticket. In this attack Bob reads a value of type nonce and confuses it for the ticket, this is called a type flaw.

The attack breaks liveness and strong liveness for both Alice and Simon, because they have not performed a single action in this trace. Another important point to note is that there is no initiation necessary for this attack, meaning that the intruder does not have to wait until one of the agents initiates a session, but is able to initiate the attack by itself. Apart from the fact that no initiation is necessary there is also no need for an untrusted agent to participate in the protocol.

2.2.3 Attack Example: Key Compromise

In order to illustrate key compromise a protocol, that also originates from Needham and Schroeder [NS78] is introduced: the Needham Schroeder Symmetric Key protocol. Note that this is a different protocol from the public key version of the protocol given in section 2.1.

This protocol has the following security goals: mutual authentication between initiator and responder and the agreement of a new fresh session key that can be used to encrypt messages between initiator and responder. Unlike the original Needham Schroeder Protocol no public key cryptography is used and agents only need to share a key with the server.

Figure 2.8 shows an attack on the Needham Schroeder Symmetric Key protocol. Here we assume that the message $\{78492, Alice\}k(Bob, Simon)$ has been

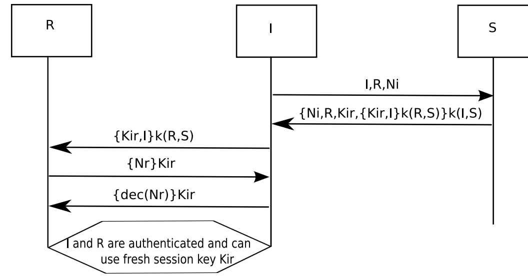

Figure 2.7: Informal Description of The Needham Schroeder Symmetric Key Protocol

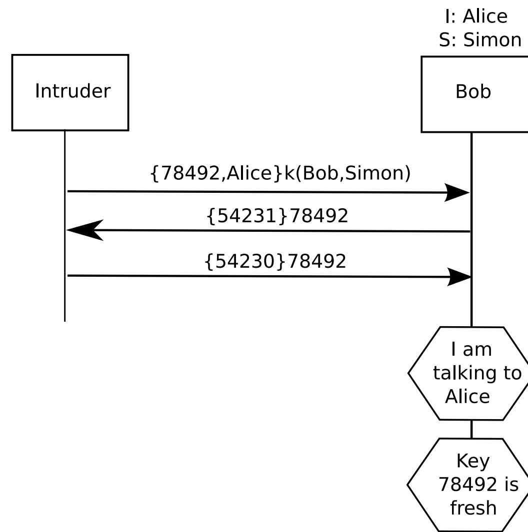

Figure 2.8: Attack on the Needham Schroeder Symmetric Key Protocol

recorded in a previous run of the protocol and that the corresponding session key 78492 has been compromised by the intruder. This information lets the intruder finish the run with Bob, who then believes that Alice is talking to him and that the old compromised key is a freshly generated session key.

The attack breaks liveness and strong liveness for both Alice and Simon, because they have not performed a single action in this trace. Another important point to note is that there is no initiation necessary for this attack, meaning that the intruder does not have to wait until one of the agents initiates a session, but is able to initiate the attack by himself. Apart from the fact that no initiation is necessary there is also no need for an untrusted agent to participate in the protocol. The attack does have an extra requirement, the fact that the intruder is able to compromise session keys.

Chapter 3

Systematic Analysis of Attacks: a Domain Analysis

The previous chapter contained a manual and non-systematic analysis of the interesting properties related to three attacks. The goal of this research is to define a more systematic approach to identify the properties that define the essence of an attack. In this chapter a domain analysis is performed in order to identify properties that define the essence of attacks. Chapter 4 formalizes these properties using the operational semantics as defined in [CM05].

We focus on three independent aspects of attacks:

- Consequences
  The consequences of attack determine the strength of the attack. The higher the consequences of an attack the more interesting it is.
- Requirements
  Apart from the consequences of an attack an attack also has requirements. These requirements can be on the environment, the protocol implementation or on the behaviour of certain agents. The lower the number of requirements the more interesting the attack is.
- Complexity
  An attack has a certain complexity, for example the number of messages that are exchanged and the number of operations that the intruder has to perform. The lower the complexity of an attack the more interesting the attack is.

In order to systematically analyze the characteristic properties that define the essence of an attack the domain analysis approach is used. We use the security protocol model as given in [CM05], that consists of six sub-models, as depicted in Figure 3.1.

In this research we assume that the protocol description and the communication model are given. For the protocol description we use the formal description method as described in [CM05] and we assume asynchronous communication with a single network buffer because this is the most general approach. The remaining four sub-models are analyzed in detail, resulting in properties associated with the sub-model in question. Properties will be listed in bold text.

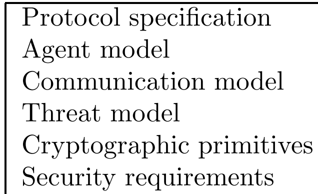

Figure 3.1: Sub-models of the Security Protocol Model

3.1 Agent Model

The formal protocol specification describes the initial knowledge of agents as well as the way the protocol should be executed. Agents execute the roles of the protocol. In general the agent model is based on a closed world assumption, meaning that the agents do not show behaviour that is not described by the protocol description. Even when we follow this assumption there still is a large degree of freedom for agents while executing the protocol. This operational freedom can be divided into two categories: freedom in how the protocol will be implemented and freedom in how the protocol will be executed. We will call the first type of freedom implementation details and the second type agent behaviour.

3.1.1 Implementation Details

It is important to realize that in order for an attack to be applicable in practice the attack has to work on a protocol implementation instead of a protocol specification. On first sight the protocol specification fully describes the operational behaviour of the protocol and can therefore be used directly to create an implementation. There are some implementation specific issues that it does not address and that are left as choices for the person that is implementing the protocol. In this research we will not address low level implementation specifics, such as the programming language or buffer sizes, but we will address two important higher level implementation decisions: type information and self sessions.

Type Information

The existing attack literature contains a large number of attacks that depend on type flaws, these attacks assume that agents that are executing the protocol are unable to distinguish the types of values that are being transmitted in the protocol. Various techniques exist to implement type checking, an example of which is described by Heather, Lowe and Schneider in [HLS00]. The choice wether or not to implement strict type checking is up to the person that is implementing the protocol.

We define the following type information related properties:

- **TypeFlawPresent**
  This property indicates wether there are type flaws present in the attack. We will not consider the use of Tickets to be type flaws, because these type flaws also occur in valid protocol executions.

- **TypeFlawOnTupleCount**
  Because values can be combined into tuples of values it is also possible for an agent to mistake a tuple of multiple values for a single value.

In general typeflaws occur because the digital representation of values of a certain type overlap with the representation of another type, for example Nonces and SessionKeys can both be considered to be random data strings and are therefore likely to be confused if no explicit type information is added. There are also data types that have a much more limited valid domain, this holds for AgentName and TimeStamp values. Even if there is no explicit type information it is unlikely that a value of a different type will be mistaken for a valid TimeStamp or a valid AgentName. This results in the following two extra properties related to type flaws:

- **TypeFlawOnTimeStamp**
  This property indicates wether the attack instantiates a variable of a different type for a TimeStamp value.

- **TypeFlawOnAgentName**
  This property indicates wether the attack instantiates a variable of a different type for an Agent value.

Self Sessions

It turns out that a significant number of theoretical attacks on security protocols use what we will be called self sessions: agents that run sessions with themselves. A protocol implementation can easily overcome these kind of attacks by explicitly disallowing self sessions. There are two different aspects to self sessions, either an agent starts a session with itself or it responds to a session that is claimed to originate from itself. We will consider the first possibility in the next section, because it is more related to agent behaviour. This leaves us with the following protocol description related property concerning self sessions:

- **SelfResponder**
  This property indicates wether the attack contains a non initiator run that contains a self session.

3.1.2 Agent Behaviour

Even when all implementation details have been sorted out agents still behave in a non-deterministic way, meaning that there is no way for the intruder to know what they will do next. In this section we try to capture the essence of this non-deterministic behaviour in properties. Therefore we analyze the points in the protocol where agents have freedom, even when they are closely following the protocol implementation. We distinguish four different types of events that give the agents this freedom: which protocols to execute, which roles to execute, when to start a new run and freedom related to initiation.

3.1.3 Executed Protocols

In the past most effort has gone into analyzing security protocols under the assumption that only one protocol is being executed by the involved agents. In

this research we will also consider attacks that require agents to run multiple protocols in parallel, we call these attacks multi-protocol attacks. We will use the following property to distinguish multi-protocol attacks:

- **InvolvedProtocols**
  The set of protocols that is involved in the attack.

3.1.4 Executed Roles

When the set of protocols that can be executed has been decided an agent can still decide which roles of the protocols it will execute. In this section we will focus purely on trusted agents, untrusted agents will be discussed Section 3.2. The number of roles and actions that are performed by trusted agents influence the attack complexity, therefore we define the following properties:

- **HonestRoleCount**
  Number of different roles for which a trusted agent has performed at least one action.

- **HonestActionCount**
  Number of send and read actions performed by honest agents in the attack.

In general we assume that every agent can perform all roles of the protocol. In a regular protocol execution agents perform a single role and server roles are only executed by servers. This yields the following two feasibility related properties:

- **MultipleRoleAgent** This property indicates wether the attack requires an agent to perform multiple different roles.

- **ServerInOtherRole** This property indicates wether the attack requires a server to perform multiple different roles.

3.1.5 Runs

Apart from the agents free choice of what roles to execute there can also be specific restraints set by the environment, for example in a smart card setting the limited computing power of a smart card can impose a limit on the number of parallel runs that an agent can execute. We call a run a parallel run if the agent that is executing it, is also involved in another run at the same time. We define the following run related properties.

- **HonestRunCount** Number of runs performed by honest agents in the attack.

- **ParallelRunCount** Number of parallel runs performed by honest agents in the attack.

- **IncompleteSessions** This property indicates whether the attack leaves behind incomplete sessions. Incomplete sessions are sessions that have not completed the last action described in the protocol description.

3.1.6 Initiation

In the semantics there is no distinction between the creation of an initiator run (which we call initiation) and the creation of responder runs. In our analysis of attacks we observe that initiation is of great importance for the feasibility of an attack, because it is the only event in a trace that the intruder can not influence or predict. All other events can be influenced by the intruder, for example by delaying or injecting messages responder runs can be created at a suitable time.

The next two sections discuss two distinct aspects of initiation: the involved parties and the moment of initiation.

Involved Parties

The parties that are involved in an initiation influence its feasibility. It is feasible that an agent initiates a session with some other agent, but it may be less feasible if an agent has to initiate a session to a specific other agent (for example the intruder). These properties are per initiation, to classify an entire attack all initiations need to be addressed.

- **RandomInitiation**
  The involved agent in a random agent.

- **SpecificInitiation**
  The involved agent is a specific trusted agent.

- **ToIntruderInitiation**
  The involved agent must be an untrusted agent.

- **SelfInitiation**
  The involved agent is a responder and it is the same as the agent performing the initiation (note that we only look at the Responders here and not at all involved agents).

Moment of Initiation

Apart from the fact that the intruder has no influence on the involved parties of an initiation, it also has no influence on the moment that the initiation will take place. By definition an initiation can take place when no other run exists because it is completely independent on other runs. If the attack requires the initiation to take place when another run is active, this means that the attack imposes a time constraint on the initiation.

- **OwnRunInitiationCount**
  Number of other runs of the agent performing the initiation that have to be active at the moment of initiation.

- **OtherRunInitiationCount**
  Number of other runs of other agents that have to be active at the moment of initiation.

3.2 Threat Model

The threat model describes the capabilities of the intruder, we distinguish three different intruder capabilities: network capabilities, untrusted agents and intruder complexity.

3.2.1 Network Capabilities

When attacks are published in literature often the Dolev-Yao intruder model is assumed, in this model the intruder has complete control over the network. It is possible that the protocol is being run in an environment that does not allow the intruder complete control over the network. An example of such an environment is wireless communication, in which it is hard to deflect a message (i.e. both eavesdrop and jam a message at the same time).

We define the following network capabilities related properties:

- **NetworkJam**
  The attack requires the intruder to block network messages.

- **NetworkInject**
  The attack requires the intruder to inject messages into the network.

- **NetworkDeflect**
  The attack requires the intruder to deflect network messages, i.e. jam and eavesdrop at the same time.

- **NetworkEavesdrop**
  The attack requires the intruder to eavesdrop on network messages.

3.2.2 Untrusted Agents

It is assumed that the intruder can conspire with agents in order to complete an attack. A famous example of this is the attack on the Needham Schroeder protocol where agent Eve conspires with the intruder in order to attack Bob. We distinguish two different kind of agents: regular agents and servers and therefore two different properties related to untrusted agents:

- **UntrustedAgent**
  The attack requires the involvement of an untrusted agent.

- **CompromisedServer**
  The attack requires an untrusted agent to perform a server role.

3.2.3 Intruder Complexity

An attack may also require an intruder to execute complex operations, for example encrypting and decrypting messages to form new messages that are required to carry out the attack. In order to analyze this complexity we introduce the following property:

- **IntruderActionCount**
  The number of internal intruder actions that have to be carried out by the intruder. If there is no intruder present in the attack this will be zero.

3.3 Cryptographic Primitives

In this analysis we will follow the so called black-box approach to cryptography, meaning that we will not consider the internal implementation of cryptographic primitives and will assume that they are perfect. We will consider one exception to the perfect cryptography assumption, namely session key compromise. We will assume that the intruder is capable of compromising short lived session keys after the session has been terminated. This results in the following property:

- **KeyCompromise**
  This property indicates wether the attack requires the intruder to compromise a session key.

3.4 Security Requirements

Security requirements define the security goals that a protocol should fulfill, in [CM05] these are expressed using local security claims. We will analyze the consequences of an attack by looking at the claim that are broken. Apart from the claims that are present in [CM05]: Secrecy, Agreement and Synchronization we also analyze the following security claims in this research: Liveness, Strong Liveness and Freshness. Per claim type we analyze the number of broken claims with a different label. This results in the following properties:

- **BrokenSecrecyCount**
  Number of broken secrecy claims with a different label.
- **BrokenSynchronizationCount**
  Number of broken synchronization claims with a different label.
- **BrokenAgreementCount**
  Number of broken agreement claims with a different label.
- **BrokenLivenessCount**
  Number of broken liveness claims with a different label.
- **BrokenStrongLivenessCount**
  Number of broken strong liveness claims with a different label.
- **BrokenFreshnessCount**
  Number of broken freshness claims with a different label.

Chapter 4

Formal Definitions of Properties

The previous chapter introduced a number of properties in an informal way. In this chapter, these properties are formalized when possible, this formalization serves as the basis for the automation in Chapter 5. The properties fall into three main categories: Consequences, Requirements and Complexity. Note that because of time constraints the properties related to intruder network capabilities have not been formalized.

4.1 Notation

All definitions are given using the formal semantics described in [CM05]. All definitions are for a given attack trace α. An attack trace is a finite sequence of actions. Individual actions are represented by a 4-tuple: $(rid, \rho, \sigma, event)$ where rid is a unique run identifier, ρ is used to represent the role to agent mapping, σ is used to represent the variable to value mapping and *event* describes the event that is performed. We will use the following notation conventions:

- We write α_i for the i-th element of trace α.
- We use the index function to obtain the index of an action in the trace: $\alpha_i = action \Leftrightarrow index_\alpha(action) = i$.
- Because all definitions are for a given trace α, we leave out the subscript alpha in the index notation.
- We abuse this notation and say that $action \in \alpha \Leftrightarrow \exists(i : \alpha_i = action)$.
- We use the $actor(action)$ method to map an action in α to the role that is executing this action.
- We assume there exists a function ProtocolName that maps an action to a protocol name, this function is considered to be an addition to the semantics defined in [CM05].

- We assume there exists a function LastRunAction that is able to determine if an action is the last action that has to be performed in a run according to the protocol specification.
- We assume that we can use the predicate Free on a variable to determine if its value is unbound, meaning that for all possible substitutions of this variable by another value of the same type the resulting trace is still valid.
- We do not explicitly mention function signatures. For input types we assume that action has type *action*, α has type *trace*, agent has type *agent* and initiation is an action for which the Initiation function returns True. All properties that are postfixed with Count have a natural number as output type and all other properties have boolean output types.

4.2 Definitions

We introduce a number of new definitions, in order to express the properties more concisely.

4.2.1 General Definitions

We use the following projections to extract specific elements from the action tuples:

- Extracting run identifier from an action:
  $runid(rid, \rho, \sigma, event) = rid$
- Extracting role definitions from an action:
  $rho(rid, \rho, \sigma, event) = \rho$
- Extracting variable substitution from an action:
  $sigma(rid, \rho, \sigma, event) = \sigma$

4.2.2 Definitions Related to Actions and Roles

- Agent executing an action
  $ExecutingAgent(action) = rho(action)(actor(action))$
- Agents that are believed to be involved in an attack
  $InvolvedAgents(\alpha) = \bigcup_{action \in \alpha} rng(rho(action))$
- Roles that are believed to be involved in an attack
  $InvolvedRoles(\alpha) = \bigcup_{action \in \alpha} dom(rho(action))$
- Runs that are involved in the attack
  $InvolvedRuns(\alpha) = \bigcup_{action \in \alpha} runid(action)$
- Roles that are believed to be fullfilled by a specific agent in an attack
  $BelievedRoles(agent, \alpha) =$
  $\{role \in InvolvedRoles(\alpha) : \exists(action \in \alpha : rho(action)(role) = agent)\}$
- Roles that are actually fullfilled by a specific agent in an attack
  $PerformedRoles(agent, \alpha) =$
  $\{role(action) : \exists(action \in \alpha : ExecutingAgent(action) = agent)\}$

4.2.3 Definitions Related to Runs

- Action a is performed before action b in an attack
  $Before(a, b, \alpha) = a \in \alpha \wedge b \in \alpha \wedge index(a) < index(b)$

- Action a is performed after action b in an attack
  $After(a, b, \alpha) = a \in \alpha \wedge b \in \alpha \wedge index(a) > index(b)$

- Runs that are active while an action is being performed
  $ActiveRuns(action, \alpha) =$
  $\{rid \in InvolvedRuns(\alpha) : \exists(a{\in}\alpha : runid(a) = rid \wedge Before(a, action, \alpha))$
  $\wedge\, \exists(b{\in}\alpha : runid(b) = rid \wedge After(b, action, \alpha))\}$

- Runs from a specific agent that are active while an action is being performed
  $ActiveAgentRuns(action, \alpha, agent) =$
  $\{rid \in InvolvedRuns(\alpha) : \exists(a{\in}\alpha : runid(a) = rid \wedge Before(a, action, \alpha))$
  $\wedge\, ExecutingAgent(a) = agent \wedge$
  $\exists(b{\in}\alpha : runid(b) = rid \wedge After(b, action, \alpha)) \wedge \}$

Definitions Related to Feasibility

- Initiation(action)
  A given action is an initiation if the action is the first event in a run and has not been preceded by any read or send events in the same run.
  $Initiation(action) =$
  $type(action) = send \wedge \neg\exists(a \in \alpha : runid(a) = runid(action) \wedge index(a) <$
  $index(action) \wedge type(a) \in \{read, send\})$

- Initiator(initiation)
  The agent that is performing a given initiation.
  $Initiator(initiation) = ExecutingAgent(initiation)$

- InitiatorRun(action)
  Indicates if the action is part of an initiator run.
  $InitatiorRun(action) = \exists(a \in \alpha : rid(a) = rid(action) \wedge Initiation(a))$

- Responders(initiation)
  The collection of agents that are instantiated for at least 1 non initiator role for a given initiation. This definition uses the assumption that a protocol only contains one initiator role.
  $Responders(initiation) = \{rho(initiation)(r) : r \in dom(rho(initiation)) \backslash role(initiation)\}$

- InvolvedAgents(initiation)
  All agents involved in the initiation.
  $InvolvedAgents(initiation) = rng(rho(initiation))$

Definitions Related to Consequences

- BrokenClaimCount
  $BrokenClaimCount(claimtype, \alpha) =$
  $|\{label(claim) : claim \in BrokenClaims(\alpha) \wedge type(claim) = claimtype\}|$

4.3 Consequences

- **BrokenSecrecyCount**
  $BrokenSecrecyCount = BrokenClaimCount(secrecy, \alpha)$

- **BrokenSynchronizationCount**
  $BrokenSynchronizationCount = BrokenClaimCount(synchronization, \alpha)$

- **BrokenAgreementCount**
  $BrokenAgreementCount = BrokenClaimCount(agreement, \alpha)$

- **BrokenLivenessCount**
  $BrokenLivenessCount = BrokenClaimCount(liveness, \alpha)$

- **BrokenStrongLivenessCount**
  $BrokenStrongLivenessCount = BrokenClaimCount(strongliveness, \alpha)$

- **BrokenFreshnessCount**
  $BrokenFreshnessCount = BrokenClaimCount(freshness, \alpha)$

4.4 Requirements

We distinguish three different groups of attack requirements, some properties would be appropriate in multiple groups. These properties are only listed once in the most appropriate group. Note that the group in which a property belongs can change with the environment in which the protocol is being used.

4.4.1 Type Flaws

- **TypeFlawPresent**
  $TypeFlawPresent =$
  $\exists(action \in \alpha : \exists(var \in dom(sigma(action)) : type(var) \neq type(sigma(action)(var))))$

- **TypeFlawOnAgentNames**
  TypeFlawOnAgentNames =
  $\exists(action \in \alpha : \exists(var \in dom(sigma(action)) : type(var) \neq type(sigma(action)(var))$
  $\wedge\ type(var) = agent))$

- **TypeFlawOnTimeStamps**
  $TypeFlawOnTimeStamps =$
  $\exists(action \in \alpha : \exists(var \in dom(sigma(action)) : type(var) \neq type(sigma(action)(var))$
  $\wedge\ type(var) = timestamp))$

- **TypeFlawOnTupleCount**
  $TypeFlawOnTimeStamps =$
  $\exists(action \in \alpha : \exists(var \in dom(sigma(action)) : type(var) \neq type(sigma(action)(var))$
  $\wedge\ type(var) = tuple))$

4.4.2 Agent Roles

- **SelfResponder**
  $SelfResponder = \exists(action \in \alpha : \neg InitatiorRun(action) \wedge$
  $\exists(role \in dom(rho(action)) \backslash actor(action) :$
  $rho(action)(role) = rho(action)(actor(action))))$

- **MultipleRoleAgent**
  $MultipleRoleAgent =$
  $\exists(agent \in InvolvedAgents(\alpha) : |PerformedRoles(agent, \alpha)| > 1)$

- **ServerInOtherRole**
  $ServerInOtherRole =$
  $\exists(agent \in InvolvedAgents(\alpha) :$
  $|PerformedRoles(agent, \alpha)| > 1 \wedge S{\in}PerformedRoles(agent, \alpha))$

4.4.3 General Requirements

- **IncompleteSessionCount**
  $IncompleteSessionCount =$
  $|\{rid \in InvolvedRuns(\alpha) : \exists(action \in \alpha : runid(action) = rid{\wedge}LastRunAction(action)\}|$

- **InvolvedProtocols**
  $InvolvedProtocols(\alpha) = \bigcup_{action \in \alpha} ProtocolName(action))$

4.4.4 Intruder Capabilities

- **UntrustedAgent**
  $UntrustedAgent = InvolvedAgents(\alpha) \cap \mathcal{A}_U \neq \emptyset$

- **CompromisedServer**
  $CompromisedServer = \exists(agent \in \mathcal{A}_U : S{\in}BelievedRoles(agent))$

- **KeyCompromise**
  $KeyCompromise = \exists(action \in \alpha : KeyCompromise(action))$

4.4.5 Feasibility

- **RandomInitiation**
  $RandomInitiationCount =$
  $|\{action \in \alpha : Initiation(action) \wedge \exists(agents \in InvolvedAgents(action) :$
  $Free(agent))\}|$

- **SpecificInitiationCount**
  $SpecificInitiationCount =$
  $|\{action \in \alpha : Initiation(action) \wedge \exists(agents \in InvolvedAgents(action) :$
  $\neg Free(agent))\}|$

- **ToIntruderInitiationCount**
  $ToIntruderInitiationCount =$
  $|\{action \in \alpha : Initiation(action) \wedge InvolvedAgents(action) \cap \mathcal{A}_U \neq \emptyset\}|$

- **SelfInitiation**
$SelfInitiationCount =$
$|\{action \in \alpha : Initiation(action) \wedge Initiator(action) \in Responders(action)|$

- **OwnRunInitiationCount**
$OwnRunInitiationCount =$
$|\{action \in \alpha : Initiation(action) \wedge \exists (run \in ActiveRuns(action, \alpha) :$
$runid(run) \neq runid(action) \wedge rho(run)(actor(run)) = Initiator(action))\}|$

- **OtherRunInitiationCount**
$OtherRunInitiationCount =$
$|\{action \in \alpha : Initiation(action) \wedge \exists (run \in ActiveRuns(action, \alpha) :$
$runid(run) \neq runid(action) \wedge rho(run)(actor(run)) \neq Initiator(action))\}|$

4.5 Attack Complexity

- **InitiationCount**
$InitiationCount = |\{action \in \alpha : Initiation(action)\}|$

- **HonestRoleCount**
$HonestRoleCount = |\{role(action) : action \in \alpha \wedge actor(action) \in \mathcal{A}_T\}|$

- **HonestActionCount**
$HonestActionCount = |\{action \in \alpha : actor(action) \in \mathcal{A}_T\}|$

- **HonestRunCount**
$HonestRunCount = |\{runid(action) : action \in \alpha \wedge actor(action) \in \mathcal{A}_T\}|$

- **ParallelRunCount**
$ParallelRunCount =$
$|\{rid(action) : \exists (action \in \alpha :$
$|ActiveAgentRuns(action, \alpha, ExecutingAgent(action)) > 1|\}|$

- **IntruderActionCount**
$IntruderActionCount = |\{action : action \in \alpha \wedge actor(action) \notin \mathcal{A}_T\}|$

Chapter 5

Experiments

Chapter 3 and Chapter 4 describe a collection of properties that describe the essential characteristics of attacks. In order to verify if this collection of properties is suitable for the goals that have been set out in Section 1.2 an experiment has been performed. This chapter describes the design, execution and results of this experiment.

5.1 Experiment Goals

The goals of the experiment are to verify the goals of the method as described in Section 1.2:

- Be able to identify if a given attack is equal to another given attack
- Structure the huge amount of output generated by automated security protocol checkers by defining an order on attacks and eliminating duplicate attacks
- Be able to filter out attacks that are not relevant in a certain environment

Figure 5.1 shows the model of the experiment that has been designed to verify the goals, this model will be explained in the next sections.

5.2 The Experiment Tool

The experiment tool is a newly developed tool written in the Python programming language, using existing open source libraries to read XML input and to cache results in a database. The tool has been designed in an object oriented and extensible manner.

The experiment consists of three phases: the analysis phase, the classification phase and the filtering and sorting phase. Filtering and sorting can be dynamically adjusted by the user, allowing the user to obtain a list of the attacks that are most interesting in a parctical environment. In order to speed up subsequent experiments on the same input set both the results from the analysis and the classification phase are stored in a database. The next sections explains the phases in more detail.

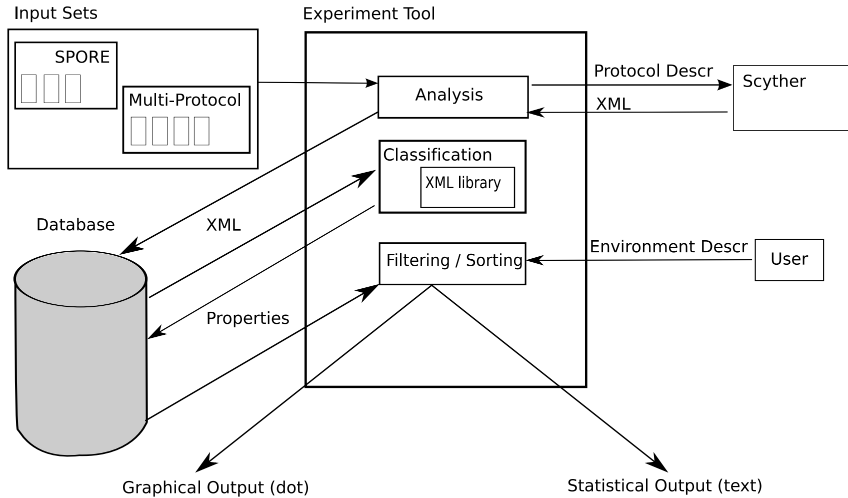

Figure 5.1: Experiment Model

5.2.1 Analysis Phase

The analysis phase takes a number of Scyther input files that together form the input of an experiment. It then calls Scyther to analyze the input files one by one, caching the XML results Scyther produces in a database so that if the same input file is analyzed more than once (for example when the experiment is repeated) it can used the cached database result instead of calling Scyther.

5.2.2 Classification Phase

When the XML output for a certain attack is known the XML library is used to parse the XML output into an internal attack representation. For this representation the value of all the properties described in Chapter 4 is determined. This list of properties is also stored in the database in order to prevent duplicate classification of the same XML input.

5.2.3 Filtering and Sorting Phase

When all properties have been assigned the resulting attacks are filtered according to filters that can be defined by the user. These filters can be based on properties or custom filtering functions implemented in the Python programming language. After the results have been filtered they are sorted according to a user specified weight for properties related to the consequences of the attack and the requirements and complexity.

5.3 Input Sets

In this research we will analyze two different input sets, both sets contain protocols from the Security Protocols Open Repository (SPORE) [SPO03]. SPORE is a freely usable repository that contains 45 different security protocols and

the corresponding known attacks on these protocols. We will look at these protocols in isolation (the SPORE input set) and at all possible combinations of two protocols from SPORE executed in parallel (the Multi Protocol input set). These two input sets will be described in more detail in the following sections.

5.3.1 SPORE

In order to analyze the Security Protocols Open Repository (SPORE) all suitable protocols were modelled as input files for Scyther, for a variety of reasons the following 6 protocols were not included:

- CAM
  This protocol only contains of a single send action.

- Diffie Helman
  This protocol uses mathematical properties that can not be modelled in Scyther.

- GJM
  This protocol uses constructions that can not be modelled in Scyther.

- Gong
  This protocol uses mathematical properties that can not be modelled in Scyther.

- Kerberos V5
  This protocol has been omitted because of its complexity.

- SK3
  This protocol uses mathematical properties that can not be modelled in Scyther.

The other 39 protocols in SPORE were modelled as closely to the description in SPORE as possible.

5.3.2 Multi Protocol

Multi protocol attacks are attacks that occur when two or more protocols are being run in parallel. In this research we restrict ourself to a maximum of 2 parallel protocols, three and more parallel protocols are considered future research.

The first paper to mention multi protocol attacks comes from Kelsey,Schneier and Wagner [KSW97], this paper only considers chosen protocol attacks, claiming that for every correct protocol a new protocol can be constructed to break the correct protocol. The first published results of practical multi protocol attacks on real world protocols is a report by Cas Cremers [Cre05b]. One of the goals of this thesis is to systematically analyze the wide variety of multi protocol attacks.

5.4 The Scyther Tool

This research will use the existing Scyther tool [Cre05a] to generate attack traces, this section describes the history of the tool, its technical details, attack representation and current limitations.

5.4.1 History and Development

Scyther has been developed at the ECSS group of the Technical University of Eindhoven as a part of the PhD research performed by Cas Cremers. Its development started in December 2003 as a model checker based on the alternative definitions of Synchronization and Agreement described in [CMdV03]. Later in August 2004 support for a second approach, called the Arachne engine was added. During the research described in this thesis Scyther was still under development.

5.4.2 Technical Details

Scyther is written in the C programming language and consists of approximately 16.000 lines of code. Attacks found by Scyther can be exported in XML, allowing them to be read by other tools. Scyther can use two different techniques to analyze security protocols: a finite state model checker and a backward symbolic state search technique. The model checker approach investigates all reachable states in the system and identifies states in which one of the security claims does not hold. The backward symbolic state search technique which will be referred to as the Arachne engine is based on the Athena method that has been developed by D. Song [Son99]. The Arachne engine finds attacks by searching backwards from the claim that is broken. This technique allows full type flaws and can explore infinite state spaces.

In this research only the Arachne engine will be used, because it is faster, more scalable and it has support for typeflaws and tickets.

5.4.3 Attack Representation

When using the Arachne engine Scyther will output attacks in semi trace format. A semi trace is a representation of a (possibly infinite) set of corresponding traces. The differences between semi traces and regular traces are described below:

- Value respresentation
  In regular traces the actual values are used to represent bound variables, for example in Figure 2.4 the number 78732432 is used to represent the nonce value generated by the initiator (Ni). In semi traces a symbolic representation is used, for example $Ni\#2$ to represent the value of Ni generated in run 2.

- Unbound variables
  In regular traces there is no distinction between bound and unbound variables, both are represented by their corresponding values. In semi traces unbound variables are post-fixed using a V, meaning that the values of

these variables can be chosen freely (for example by the attacker) as long as it is done consistently.

- Ordering of events
  Regular traces define a strict ordering of events, semi traces can allow for multiple orderings because ordering in semi traces is described using partial ordering. In semi traces this ordering is described by making the events nodes in a directed graph, that are enabled when all nodes that are connected the action have been enabled.

The figures below give an example of a semi trace and two different corresponding regular traces. In this figure arrows denote the partial ordering and the role assignment is given inside the diamond shape at the top of the run.

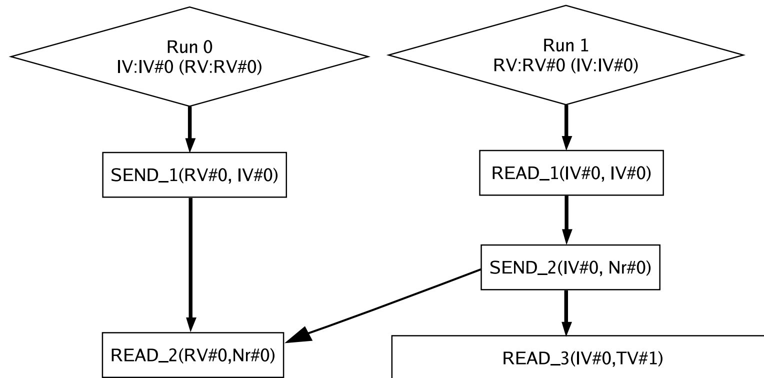

Figure 5.2: Example of a Semi Trace

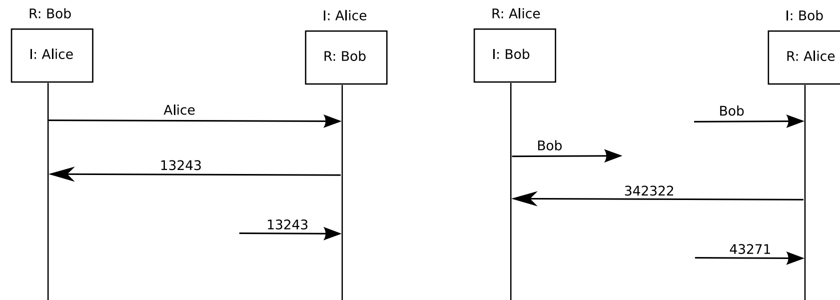

Figure 5.3: Two Different Traces Corresponding to the Same Semi Trace

- Because $IV\#0$ and $RV\#0$ are unbound variables different agent names can be filled in for them, meaning that different agents can fulfill the roles of initiator and responder. In the first regular trace Alice is filled in for $IV\#0$, in the second trace Bob is.

- Because there is no ordering defined on READ_1 and SEND_1, READ_1 can be after SEND_1 as depicted in the first trace or before SEND_1 as depicted in the second trace.

- READ_3 has no corresponding SEND_3 event, meaning that this value is injected into the network by the intruder.

- Because the value $TV\#1$ is unbound the intruder can fill in any value for this ticket, for example the Nonce generated by the responder as depicted in the first trace or a completely new value that the intruder can construct as depicted in the second trace.

5.4.4 Current Limitations of The Scyther Tool

Unfortunately the Scyther tool also has a few limitations that are important to note for this research.

Limited Set of Claims Supported

Currently Scyther only supports three different security claims: Non Injective Synchronization, Non Injective Agreement and Secrecy. In order to analyze the consequences of an attack we would prefer to able to address all claims described in Section 2.1.4.

One Broken Claim Per Attack

Apart from the fact that Scyther only supports a limited subset of the security claims that this research tries to address it also contains no support to find attacks that break multiple claims. This also limits the possibilities of analyzing the consequences of an attack.

Key Compromise

Apart from the network capabilities that the Dolev-Yao intruder model provides the intruder is also assumed to be capable of compromising session keys, Scyther currently has no built-in support for key compromise.

5.4.5 Working Around Current Limitations

The limitations mentioned in the previous subsections need to be addressed in order to come to a useable analysis of attacks. In order to overcome the limitations the following techniques have been used:

Limited Set of Claims

In order to overcome the problem of the limited claim set we added ignored claims to Scyther, this are claims that will be ignored by Scyther but are present in the trace. By analyzing the claim status of these claims outside of Scyther we can find attacks that break claims that are not supported by Scyther itself. Note that this technique does not guarantee that Scyther finds all possible attacks for these claims because it is not explicitly searching for them.

Multiple Broken Claims

In order to obtain attack traces that can potentially contain multiple broken claims extend methods have been added to Scyther. These extend methods will try to finish incomplete runs in a trace, thereby reaching multiple claims. By combining these extend methods with the claim analysis outside of Scyther we can discover multiple broken claims in a single attack semi trace.

Key Compromise

The key compromise problem can be solved by explicitly modelling key compromise in the input files. This is done by creating an additional internal protocol that discloses session keys after a session has expired. This technique is described in more detail in Section 6.1.

5.5 Experiment Execution

The results mentioned in the next section are from an experiment that was performed on a 3GHz Intel Pentium 4, running the Fedora Core 4 Linux distribution. All these experiments were performed using Scyther revision 1369. Both input sets were analyzed without typeflaws and allowing full type flaws (Scyther -m0 and -m2 switches), all Scyther runs were limited to four parallel runs and a maximum of 200 attacks per input file. For these experiments Scyther attack pruning was disabled, meaning that all attacks that are found are reported in the XML output instead of just the shortest attacks.

5.5.1 Performance

In total Scyther generated 4.9 Gigabytes of XML output for the experiments described in Section 5.5, in total there were 115,000 attack semitraces, that were analyzed in 9.5 hours. Resulting in a speed of approximately 3.3 attacks per second. No performance tuning has been performed and it is believed that a higher performance can be obtained, for example by re-implementing some portions of the code in a lower level language.

5.6 General Results

5.6.1 Identifying Identical Attacks

By looking at the number of attacks that Scyther produced and the number of unique property assignments, we see that over 85% of the attacks that are

generated are duplicates with respect to the properties defined in Section 3. A manual analysis of a sample of these equal attacks indicated that this method is suitable for eliminating duplicates.

5.6.2 Structuring Large Collections of Attacks

The method is very useful in structuring large collections of attacks. The attacks can be sorted according to scores defined by the user. This has allowed us to find specific interesting attacks in the large multi protocol input set that would have not been found without this automated method, see Section 5.8 for more details.

5.6.3 Filtering Attacks

The high number of attacks shows that there are a lot of attacks that are not interesting, examples of this are small variations of existing attacks, very infeasible attacks and so called ticket attacks. Ticket attacks are attacks on protocols that contain a ticket, because the agent that receives the ticket simply forwards it to another agent without inspecting its contents it is possible for the intruder to inject a different value for the first occurrence of the ticket and the correct value for the forwarded version. This attack breaks agreement and synchronization but is not considered to be very interesting. Unfortunately our current definitions of agreement and synchronization are broken in a situation that only ticket attacks are present. We consider adapting these definitions or adding new properties to detect ticket attacks as future research, see Section 7.1. Other uninteresting attacks can be easily filtered out based on the properties.

5.7 Results Specific to SPORE

5.7.1 Without Type Flaws

This input set consists of 1098 attacks, of which 502 have a unique classification based on the properties in Chapter 3. All attacks break at least one synchronization claim, meaning that there are no protocols that do not satisfy secrecy for a passive intruder. Many attacks require initiation (95%) or an intruder insider (66%). If we remove infeasible attacks we obtain a collection of 160 attacks. This collection contains all attacks in SPORE, except for those against injective claims, containing type flaws and attacks that are not related to security claims. There are also new attacks, these are described in more detail in Section 6.2.1.

5.7.2 With Full Type Flaws

If we allow full type flaws we find 712 more attacks, of which 204 have a unique classification. We see that in this collection of attacks there are far more attacks that break secrecy and (strong) liveness. Indicating that the consequences of type flaw attacks are larger than those of non typeflaw attacks for the protocols in SPORE.

If we remove all infeasible attacks we obtain a collection of 37 new feasible type flaw attacks on SPORE. Of these 37 attacks a number of attacks are not present in SPORE, these are described in more detail in Section 6.2.1.

5.8 Results Specific to Multi Protocol

5.8.1 Without Type Flaws

The multi protocol set is larger, we find 6971 multi protocol attacks without typeflaws, of which 2920 have a unique classification. This set contains a lot of attacks on claims that are already broken in the single protocol setting therefore we filter out these attacks, this leaves us with 511 attacks that break claims that were correct in the single protocol setting. We observe that there is a protocol that is present very often in these attacks: Woolam-Pi and variants (97%).

Closer inspection of the attacks containing Woolam-Pi or variants shows us these protocols contain an encryption oracle for the key shared between responder and server. Because in step 4 the responder blindly encrypts a received ticket together with the name of the initiator with the key that is shared between responder and server. This allows for a large number of protocols that use symmetric keys to be broken, because the ability to encrypt a message with a certain key is not limited to the owner of that key any more, but can be achieved by sending the message to a Woolam-Pi responder.

If we remove all attacks generated by the Woolam-Pi oracle we obtain a collection of 14 attacks, all of which break a synchronization or agreement claim by exploiting the similarities between protocols and their variants, for example a regular yahalom server can complete a run between a yahalom-BAN responder and initiator. This breaks synchronization, because the labels of the send and receives of the server messages do not match, but this is merely a theoretical problem.

5.8.2 With Full Type Flaws

Allowing full type flaws in the multi protocol setting yields 22,084 new attacks, of which 7,159 have a unique classification. If we limit the set to claims that were not broken in type flaw less multi protocol setting we obtain 1,157 new attacks. Again Woolam is present in a large percentage of the attacks, but not nearly as much as before (45%).

Because we already acknowledged the dangers of using Woolam in multi protocol settings and the existence of variant only attacks, we will remove them from this analysis. We also remove attacks that use self initiation or contain typeflaws on timestamp values and typeflaws on tuple count of unencrypted values, because this is believed to be highly infeasible. This leaves us with 27 multi protocol attacks of which 11 are considered to be feasible, these attacks break the previously secrecy claims in the Yahalom and Andrew-Concrete protocols, that are correct in isolation. Two new multi-protocol attacks are described in Section 6.2.2.

Chapter 6

Miscellaneous Results

While performing this research a few results were obtained that are not directly related to the analysis of security protocols in general. This chapter describes two of these results: an implementation of passive key compromise and a number of newly discovered attacks that have been found by systematically analyzing the large data sets obtained in the experiments.

6.1 Passive Key Compromise

When reasoning about the security of protocols it is often assumed that the attacker is able to obtain short lived sessions keys within a reasonable amount of time. Because of this assumption, we need an extra security requirement namely that the compromise of a session key should not have effects on the security claims of future runs of the protocol. This section describes how passive key compromise can be modelled without the need for explicit tool support.

The approach taken here does not address the fact that the intruder could have influenced the initial session, for example by injecting or changing messages in the protocol invocation to which the compromised key belongs. Therefore we will assume that the intruder can only compromise keys of previous sessions that have been executed according to protocol specifications. Resulting in a weaker version of key compromise that will be called passive key compromise.

A way to model passive key compromise is to add all messages belonging to a previous execution of the protocol and the corresponding session key to the initial knowledge of the intruder. This approach is not suitable for use by Scyther, because it is not possible to add instantiated values to the knowledge of the intruder. It also introduces an arbitrary limit on the number of session keys that have been compromised and it makes it harder in the classification to identify if a compromised key has been used.

In order to overcome these problems we model key compromise as a separate protocol with a special name namely the original protocol name postfixed with the string KeyCompromise. This protocol is then modeled as a two step protocol where first the names of involved agents are read and then a complete session between these agents, consisting of all the individual messages that would have been sent in a regular invocation of the protocol followed by the session key are sent out and thus added to the intruder knowledge.

Note that we will only consider the existence of KeyCompromise protocols when assigning properties to the attack (it will not have an influence on other properties such as number of protocols)

6.2 Newly discovered Attacks

6.2.1 Single Protocol

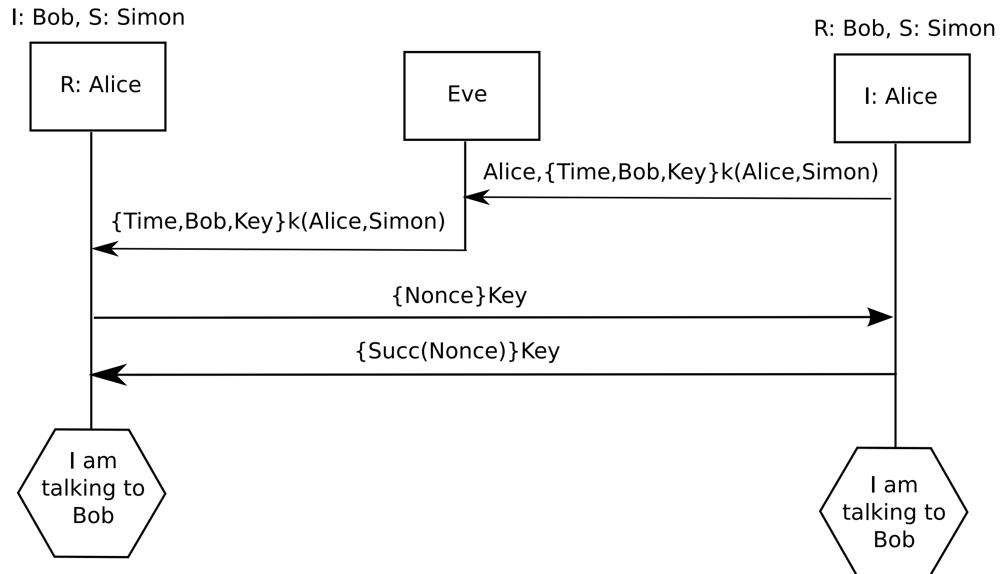

Figure 6.1: New attack on WMF-Lowe

SPORE contains an attack by Lowe on the original Wide Mouthed Frog (WMF) protocol, which results in a fixed version by Lowe (WMF-Lowe). Our analysis has yielded an attack on this protocol that is not described in SPORE. In this attack the liveness of Simon and Bob is broken.

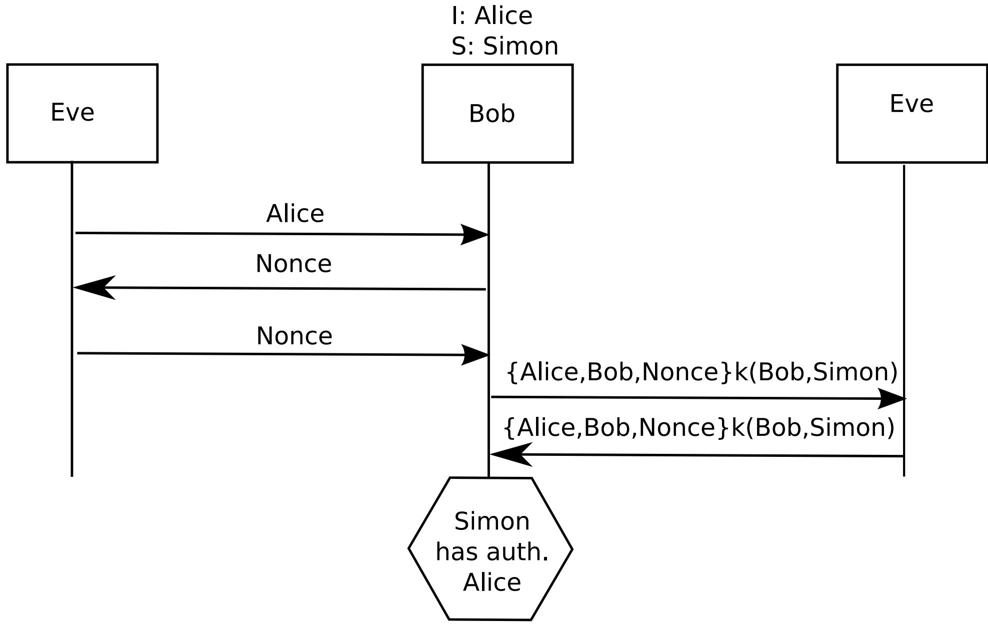

Figure 6.2: New attack on WoolamPi-1

SPORE contains no attacks on the woolamPi-1 protocol, we find an attack that utilizes a typeflaw in which the responder mistakes a nonce for a ticket. The applicability of this attack is dependant on the implementation: the responder has to forward the ticket without checking its type, which is possible because it is encrypted with a key that the responder does not know and thus difficult to verify.

6.2.2 Multi Protocol

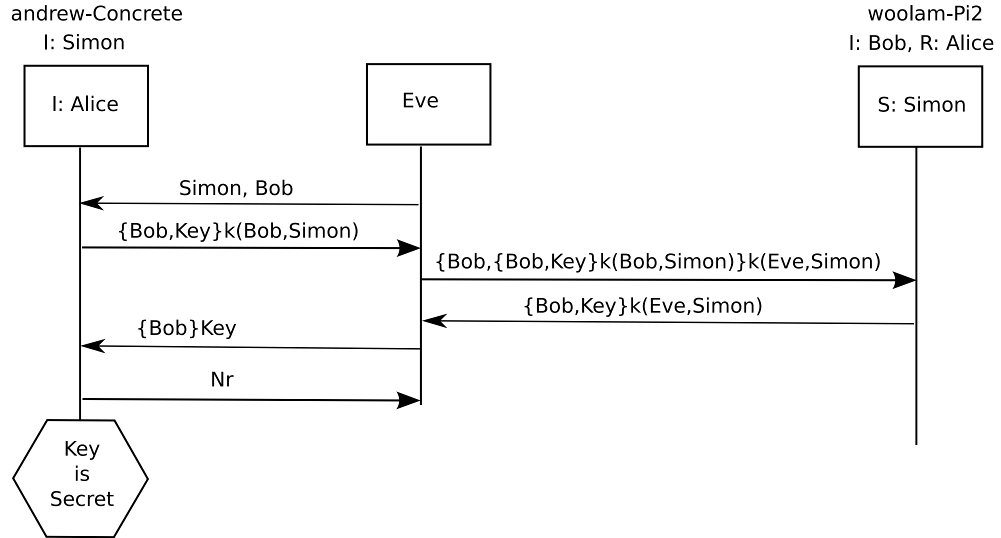

Figure 6.3: New multi protocol attack on Andrew-Concrete and WoolamPi-2

Figure 6.3 shows a multi protocol attack on the secrecy claim in the andrew-Concrete protocol when it is combined with the WoolamPi-2 protocol. In this attack there is a typeflaw in the first message where the initiator mistakes the name Bob for a Nonce.

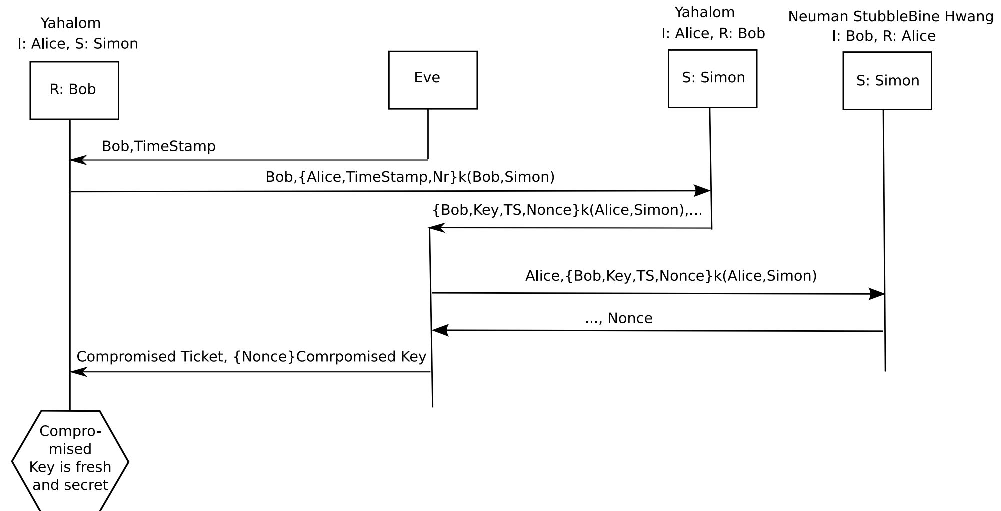

Figure 6.4: New multi protocol attack on Yahalom and Neuman Stubblebine

Figure 6.4 shows a multi protocol attack on the secrecy claim in the Yahalom protocol when it is combined with the Neumann Stubblebine Hwang protocol.

In this attack there is a typeflaw in the first message where the initiator mistakes a time stamp for a Nonce. In the attack certain irrelevant parts of the messages have been replaced with ellipses to simplify the representation. In the final step Eve sends a ticket that has been compromised in a previous session and is able to prove its validity by encrypting the Nonce (that has been leaked through the Neumann Stubblebine Hwang protocol) with the corresponding compromised session key.

Chapter 7

Conclusion

Developing the analysis method for security protocols has brought some new insights into the attack properties that are considered to be interesting, for example initiation. The automation of the process has provided insight into current limitations of the Scyther tool and has allowed analysis of a very large set of attacks, for example in the case of the multi-protocol attacks 11 interesting attacks have been found in a collection of over 20,000 attacks a result that would have been very difficult to obtain using manual analysis of the attacks. The classification method is also useful to eliminate duplicate attacks, instead of the current trace length pruning.

The next two sections contain suggestions for further research and notes about the practical applicability of this research.

7.1 Future Research

- Extend properties to include properties that describe the design of the protocol
  By adding properties that are not attack dependent, but describe the design choices of protocols, the relation between design decisions and attacks can be analyzed. An example of such a property would be a property that indicates if the protocol contains two parties that never communicate directly. By relating properties that describe the protocol design to properties that describe the consequences of an attack a greater insight into the reasons behind security attacks might be obtained.

- Formalize and implement the intruder network capabilities properties
  Because of time constraints these properties have not been formalized and implemented in this research. Adding these properties would provide a more complete classification.

- Expand the set of analyzed protocols
  There are more resources that contain collections of protocols, for example the book by Boyd and Mathuria [BM03] and other web resources. By expanding the list of protocols more attacks can be found and new interesting properties can be discovered.

- Finding a way to filter out 'Ticket Attacks'
  In the SPORE input set quite a lot of Ticket Attacks occur, these are attacks that only influence the agreement on the value of a ticket between the party that will use the ticket and the party that has forwarded the ticket. These attacks are not considered to be very interesting, therefore it would be interesting to have a way to filter them out. Strictly speaking this is a problem of the definitions of Synchronisation and Agreement that may be to strong in the ticket attack situation.

- Resolving Scyther limitations
  By using the work around methods described in Section 5.4.5, a large number of attacks that are in practice unsupported by Scyther can be found. It is believed how ever that there exist attacks that are not found this way, for example certain attacks that break multiple claims at once or attacks that violate freshness without using key compromise. If Scyther were to be extended in the future to resolve these limitations new attacks can be found.

7.2 Practical Applicability

This section deals with the practical applicability of this research and of the formal analysis of protocols in general. If we observe security issues that are reported on various internet resources we see that the vast majority of these problems do not arise from bad protocol design, but from bad implementation. Some of the properties in this research can be considered to be implementation specific, for example an attack containing a 'Self Responder' is only possible if this situation is not detected and rejected by the implementation of the protocol. In fact most attack constraints are implementation dependent.

This research can help to better understand the constraints and therefore the requirements of attacks, making it easier to assert if a certain attack can be applied to a practical system. An attack that requires parallel sessions for example might not be applicable to a smart card system.

Formal analysis of protocols is very useful, because an implementation can only be correct if the underlying protocol is correct. If the technique of proven theoretical correctness is combined with automatic program derivation, as is done for example in the work of Didelot and Lowe [Did03] a lot of current security problems could be avoided.

Bibliography

[BAN96] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. In *William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press*. 1996.

[BM03] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment*. Information Security and Cryptography. Springer, 2003.

[CM05] C. Cremers and S. Mauw. Operational semantics of security protocols. In *Scenarios: Models, Algorithms and Tools (Dagstuhl 03371 post-seminar proceedings)*. LNCS, 2005.

[CMdV03] C. Cremers, S. Mauw, and E. de Vink. Defining authentication in a trace model. In F. Martinelli T. Dimitrakos, editor, *FAST 2003*. IITT-CNR technical report, 2003.

[Cre05a] C. Cremers. Scyther documentation, 2005. `www.win.tue.nl/~ccremers/scyther`.

[Cre05b] C. Cremers. Verification of multi-protocol attacks. Computer science report csr-05-10, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2005.

[Did03] Xavier Didelot. COSP-J: A compiler for security protocols, 2003. `http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Casper/COSPJ/secu.pdf`.

[DY83] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

[HLS00] Heather, Lowe, and Schneider. How to prevent type flaw attacks on security protocols. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[KSW97] J. Kelsey, B. Schneier, and D. Wagner. Protocol interactions and the chosen protocol attack. In *Security Protocols Workshop*, pages 91–104, 1997.

[Low95] G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[Low97] G. Lowe. A hierarchy of authentication specifications. In *PCSFW: Proceedings of The 10th Computer Security Foundations Workshop.* IEEE Computer Society Press, 1997.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[Son99] D. Song. Athena: A new efficient automatic checker for security protocol analysis. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop.* IEEE Computer Society Press, 1999.

[SPO03] Security protocols open repository, 2003. `http://www.lsv.ens-cachan.fr/spore`.

[WL94] T. Woo and S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 28(3):24–37, 1994.