Master's Thesis
Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Sectie: Programmatuur
Afstudeer docent: Paul Klint
Stage begeleider: Sjouke Mauw

Model checking secrecy in security protocols

Ingmar Schnitzler
<ingmar@caribbe.an>

Amsterdam, 19 September 2003

Contents

1 Abstract **4**

2 Introduction **4**

3 Acknowledgments **5**

4 Model-checking secrecy **6**
- 4.1 The semantics behind the model 6
- 4.2 Model-checking 10
- 4.3 Requirements 11
- 4.4 Limitations 12

5 SPDL: Security Protocol Description Language **12**
- 5.1 SPDL requirements 12
- 5.2 Expressing in SPDL 13
  - 5.2.1 Functions 14
  - 5.2.2 Inverses 15
  - 5.2.3 Variables 15
  - 5.2.4 Constants 15
  - 5.2.5 Role 16
  - 5.2.6 Knowledge 16
  - 5.2.7 Definitions 16
- 5.3 The grammer of SPDL 18
  - 5.3.1 Needham-Schroeder public-key protocol SPDL example 19

6 Scenarios **21**
- 6.1 The BNF of scenario language 22
  - 6.1.1 Needham-Schroeder public key scenario example . . . 22

7 Design & implementation **23**
- 7.1 The algorithm 23
  - 7.1.1 Crippled Kao Chow Version 1 23
  - 7.1.2 The checkSecrecy algorithm 27
  - 7.1.3 The getAllPossibleInjections algorithm 29
  - 7.1.4 The fillContext algorithm 30
- 7.2 Modules 31
  - 7.2.1 Dependency Tree 32
  - 7.2.2 Knowledge 33
  - 7.2.3 Roles and Runs 34
  - 7.2.4 Functions 40
  - 7.2.5 ConfidentialityChecker 41
  - 7.2.6 Controller (Ccheck) 43
- 7.3 The initialization process 44

8 Validation 45

9 Evaluation 46

10 Future work 48

11 Learning points 48

12 Conclusion 48

13 Appendix A 49

13.1 Needham-Schröder public key protocol 49
13.2 TMN protocol 50

1 Abstract

Secrecy is an important security property in many distributed computer applications. In this Master's Thesis I present a secrecy model-checker, based on formal semantics for security protocols. I describe the language SPDL which has been created to express security protocols to this model-checker and present an evaluation of this model-checker.

2 Introduction

Security is about protecting assets and security protocols are a means to that end. Talking about security means talking about adversaries. On one side you have the secure system and on the other hand the intruder. The intruder has one objective, which is to penetrate the security that has been set up by the system.

Consider an adapted version of the Byzantine generals problem [5]. Two Byzantine armies are besieging an enemy city. This city lies in a valley, while the two armies, led by two generals, are camping on the opposite mountain ridges. The city is heavily fortified. Only a simultaneous surprise attack of the two armies would guarantee Byzantine victory. The two generals must get consensus on the time to attack the city. The problem is that they can only use messengers to get that consensus. These messengers must cross the valley, sneak through the city and climb the ridge to the other armies base camp. If captured, messengers run the risk of torture, even death, and might disclose any information they know. Not knowing that this is an undecidable problem, the generals are communicating with each other on how to get consensus. They have designed a system (long before this war) to communicate with each other, using messengers who can never provide the enemy with useful information. In order to achieve this, the sending general sends a messenger, who only carries a message containing parts of the words in the actual message. This partial message, by itself, is worthless to anyone, since it will only be readable when all the parts are combined. When the messenger returns from a successful delivery he will receive another part of the message and will try to deliver it to the receiving general. If at least the first message was sent successfully, then this process is repeated until the complete message has been sent, or the messenger is caught and killed. If the messenger is killed while carrying the first part of the message, after some time, the sending general will divide the message into new parts and send it again. If the messenger is killed during delivery of subsequent parts, the sending general will send a new messenger with the same message part and repeats this process until the complete message has been sent.

Intuitively this feels secure, if indeed the first message was sent successfully. However, great many lives depend on the secrecy of the communication be-

tween the generals. They would like to know if their security protocol is indeed secure. In order to figure this out, one general starts assessing what the city is capable of doing. Despite not being able to read complete messages, the city could send one of its own messengers. This messenger, dressed in a Byzantine uniform, would carry fake information. This way, the receiving general will regard a message from the city as a secret message from the other general. Shocked, the general learns that he might have been making decisions based on compromised information.
The Byzantine example illustrates what this project is about. Given a security protocol, a verdict is needed, on the question, whether or not it is secure. The security group of the university of Eindhoven has been working on formal semantics for security protocols [1]. These semantics will provide a general framework, which can be used to formally validate security protocols. All the work until now has been purely theoretical. The object of this project was to find out if, by using these formal semantics, a practical software application could be built which could validate security protocols. This model-checker would receive a security protocol as input and provide a verdict on the question whether or not a certain security goal has been met. Since building a model-checker for the entire semantics would be to broad a scope for a graduation thesis, I've selected a subset of these semantics by choosing one security goal, namely secrecy.
Using the semantics, a couple of protocols were checked by hand, in order to find out what kind of algorithm a human computer would use. This led to the design of the algorithm outline. By working out every detail of that outline, an implementable algorithm was designed. Implementation was done bottom-up, written in Java and using a prototyping engineering approach.
This document starts with a brief introduction to security protocols, followed by an overview of the model which will be checked, the implementation of the model-checker and ends with an evaluation of the constructed model-checker. The contributions of this work are:

- Design of the language SPDL for describing security protocols.
- Implementation of a model-checker for secrecy.
- Evaluation of this model-checker by using protocols from SPORE[1].

3 Acknowledgments

I want to take this opportunity to thank dr. Sjouke Mauw, prof. dr. Paul Klint and ir. Cas Cremers for their support in helping me building the model-checker and writing this thesis. Furthermore I'd like thank my parents Inge Boutier and Herman Schnitzler who have always been there for me.

[1]SPORE, Security Protocol Open Repository, is a public repository containing many protocols. http://www.lsv.ens-cachan.fr/spore/

4 Model-checking secrecy

In order to get an overview of the problem, a brief introduction to the semantics for security protocols is needed. These semantics are currently in development and this introduction has been derived from the uncompleted draft of the semantics [1]. Because of this, it is unavoidably incomplete. In addition, this overview is informal in nature, since complete formal discussion would be outside the scope of this graduation thesis. For more details I refer to the developing semantics.

4.1 The semantics behind the model

A protocol is a specification of message communication between two or more agents. A message consists of terms. There are three basic terms:

- *Agents*. A term identifying participants in the protocol.
- *Nonces*. A nonce is a generated number which has a unique value in the protocol. Nonces can be used to ensure freshness of a message. Consider agent A sending a message M, which includes a nonce n and is encrypted. Later in time A receives an encrypted message M', including that same nonce. Agent A can then be sure of two things; The agent that send the message M' has been able to decrypt message M and message M' has been created later in time than M.
- *Keys*. A term that can be used to encrypt other terms.

With these basic terms other terms can be constructed.

- *EncTerm*. Notation: $\{term_1\}term_2$, which indicates that $term_1$ has been encrypted with $term_2$.
- *TermList*. Notation: $(term_1, ..., term_n)$, which is an ordered list of terms.

A security protocol is a protocol which is specifically designed to guarantee certain security goals. One can classify security goals in three rough categories which are often referred to as C.I.A. [3]

- **C**onfidentiality, information which is claimed to be confidential may never be known by the intruder.
- **I**ntegrity, information which is claimed to maintain integrity may never be altered by the intruder.
- **A**vailability, information which is claimed to be available may never be withheld by intruder actions.

Within the confidentiality category there many properties can be studied. For example privacy, anonymity and secrecy. This model-checker verifies the secrecy property.
If one or more terms are claimed to be secret, then those terms may never be known to the intruder.

The semantics for security protocols use the Dolev-Yao intruder model [2]. In this intruder model, the intruder is in complete control of the network and can thus attempt to execute, what is called, a man in the middle attack. The Dolev-Yao intruder model is widely accepted at this moment. The semantics for security protocols use a model for the network in which every message sent might be caught by the intruder. It is said that an intruder learns a message when it is sent by an agent. Everything an intruder learns is stored in the intruder knowledge ($K_{intruder}$). $K_{intruder}$ is closed, meaning that for every message $\{d_1,...,d_n\}$ in the $K_{intruder}$ it also contains $f(\{d_1,...,d_n\})$. (For appropriately typed selected functions f). A function in our context could be encryption. This means that if an intruder has captured a message $\{t\}k$, a term, t, encrypted with the key, k, the intruder will add $\{t\}k$ to $K_{intruder}$ if the intruder does not know k. It will add t to $K_{intruder}$ if the intruder does know k. Further if the intruder captures a key, it will add that key to the $K_{intruder}$ as well as the contents of any message already in $K_{intruder}$ which can be decrypted with k. If an intruder knows some key k, it can construct any message $\{t\}k$ for any term, t, that can be constructed from $K_{intruder}$. The intruder can send any message, which the intruder can construct from $K_{intruder}$.
Keys can be parameterized as functions. For example if there is an agent A, it can have a public key from the function $PK : agent \rightarrow key$. In this example PK is a function that has one argument, of the type agent, and it will produce a key. So agent A's public key from the function PK can be expressed as $PK(A)$. In this model-checker two types of keys can be used; A symmetrical key and a key pair. These two key types are used to perform two different encryption types:

- Symmetrical encryption, where encryption and decryption is done with the same key. (if f is a key then $f(f(t)) = t$).
- Asymmetrical encryption, where the encryption and the decryption is done by separate keys, which are each other's inverse. (if f and f' are each other's inverse then $f'(f(t)) = f(f'(t)) = t$.).

The public-private key architecture is an example of asymmetrical encryption.
A protocol can contain many roles. A role is defined as the specified behavior of some agent. Behavior of a role is defined as a sequence of actions, will

always be executed in the right order by honest roles.
There are three types of actions a role can perform.

- *send*. An agent can send a message to another agent.
- *read*. An agent can read a message sent to it.
- *claim*. An agent can claim that a certain security goal is met at some point in the protocol. This model checker only supports claims with regard to secrecy. Terms included in a secrecy claim may never become part of the intruder knowledge.

Consider agent A sending a message M to agent B, while B is expecting a message M' from A. The intruder can do two things:

- An injection, inject a message which matches the description of M', i.e. message M from $K_{intruder}$ to agent B.
- A deflection, keep the message M from reaching agent B.

Consider, for example, a protocol[2] consisting of two roles; The role Initiator, which initiates communication and the role Responder, which responds to the initiator. This is an example protocol, where two agents, who know each others public key, share secrets trough communication. Their behavior, which consists of a sequence of actions, can be defined as follows:

$Initiator : A(B) =$
$\quad send(B,\ \{a, n_a\}publicKey(b));$
$\quad read(B,\ \{n_a, X\}publicKey(a));$
$\quad send(B,\ \{X\}publicKey(b));$
$\quad claim(secret,\ (n_a, X));$

$Responder : B(A) =$
$\quad read(A,\ \{a, Y\}publicKey(b));$
$\quad send(A,\ \{Y, n_b\}publicKey(a));$
$\quad read(A,\ \{n_b\}publicKey(b));$
$\quad claim(secret,\ (n_b, Y));$

The action $claim(secret, term\ t)$ at a specific point in the protocol means that the intruder will never learn t after that point.
It can be observed in the example protocol that, without intruder interference, each role send action is exactly matched by the other roles read action.

Role definitions only form the blueprint for the behavior of agents. A network consists of a number of parallel running agents each executing any

[2]This is not a real protocol, its only chosen to demonstrate role definitions.

number of parallel runs. A run is an instantiation of a role definition. A run can be identified by its run identifier ($runId$). Any term created by a run, inherits that run's $runId$. In the semantics for security protocols this is expressed by adding # followed by the $runId$ to the term or role which has been instantiated.
Actions are executed asynchronously. While sends and claims are non-blocking, reads are blocking. A run will therefore execute a sequence of send and claim actions until it reaches the end of the protocol or a read action is encountered. When a read is executed the run must wait until a message arrives.

The following is an example of two instantiated roles. The run Initiator, identified by the run identifier rid_1 and played by agent A communicating with agent B. The run Responder, identified by rid_2 played by agent B and communicating with A. In this example, n_a and n_b are nonces, A and B are agents, $publicKey$ is a function of type $publicKey : agent \rightarrow key$, while X and Y are variables.

$Initiator : A(B)\#rid_1 =$
 $send(B,\ \{a, n_a\#rid_1\}publicKey(b));$
 $read(B,\ \{n_a\#rid_1, X\}publicKey(a));$
 $send(B,\ \{X\}publicKey(b));$
 $claim(secret,\ (n_a\#rid_1, X));$

$Responder : B(A)\#rid_2 =$
 $read(A,\ \{a, Y\}publicKey(b));$
 $send(A,\ \{Y, n_b\#rid_2\}publicKey(a));$
 $read(A,\ \{n_b\#rid_2\}publicKey(b));$
 $claim(secret,\ (n_b\#rid_2, Y));$

Each term instantiated within a run will inherit that run's run identifier. This makes it possible to have many runs of the same role execute concurrently. Since all run identifiers are unique, the agent controlling Run rid_1 knows if a term $n_a\#rid_1$ received is the same term as the term $n_a\#rid_1$ sent. The model-checker only uses the formal name of an element, element#runidentifier to make output interpretable. All terms are implemented in the actual model-checker as Java objects. Therefore the programming environment implicitly keeps track of object origins.
A trace of a model is a list of actions from the first communication till the last. Depending on the model, many traces can exist. A trace representing successful completion of this protocol can be presented as the following sequence of actions:

 $send\ A \rightarrow B : \{a, n_a\#rid_1\}publicKey(b);$
 $inject\ A \rightarrow B : \{a, n_a\#rid_1\}publicKey(b);$

$send\ B \rightarrow A : \{n_a\#rid_1, n_b\#rid_2\}publicKey(a);$
$inject\ B \rightarrow A : \{n_a\#rid_1, n_b\#rid_2\}publicKey(a);$
$send\ B \rightarrow A : \{n_b\}publicKey(b);$
$claim\ A : secret(n_a\#rid_1, n_b\#rid_2);$
$inject\ B \rightarrow A : \{n_b\#rid_2\}publicKey(b);$
$claim\ B : secret(n_b\#rid_2, n_a\#rid_1);$

The action inject represents a term which is injected to a run by the intruder masquerading as the expected sender.

If the intruder knew the agents A and B, the public keys of agents A and B and some nonce i, then one trace representing an unsuccessful breaching attempt would be:

$send\ A \rightarrow B :\ \{a, n_a\#rid_1\}publicKey(b);$
$injec\ A \rightarrow B :\ \{a, i\}publicKey(b);$
$send\ B \rightarrow A :\ \{i, n_b\#rid_2\}publicKey(a);$

4.2 Model-checking

Model-checking is an approach to automatic verification of systems. A model-checker is a procedure that decides whether a given structure M is a model for a logical formula ϕ. In this model-checker ϕ is: $t \in C \Longrightarrow t \notin K_{intruder}$
Where t is a term, C is a set of terms which are claimed secret and $K_{intruder}$ represents the set of terms which can be derived from the intruder knowledge.

Model-checking typically depends on a discrete model of a system. The system's model can be represented by a graph structure, in which the nodes represent states. There are two ways in which the model-checking problem can be specified.

- Global model-checking problem
  Given a finite model structure, M, and a formula, ϕ, determine the set of states in M that satisfy ϕ.

- Local model-checking problem
  Given a finite model structure, M, a formula, ϕ, and a state s in M, determine whether s satisfies ϕ

The model structure checked by this model-checker consists of parallel executing runs. The model structure grows exponentially with the number of runs. This problem is called the state-explosion problem. This model-checker is implemented as a local model-checker, which helps to fight the

state-explosion problem. Scenarios are used to specify which states needs to be checked.

4.3 Requirements

This section explains the software requirements for the model-checker. Since the object is to demonstrate a practical implementation of the semantics for security protocols for security protocols, a minimal set of requirements is chosen, which could still provide a workable solution.

- The model-checker should analyze the model and produce a verdict on whether or not claims have been refuted.
- There must be an input language and a parser and type-checker must be created for that language.
- The model-checker should use the Dolev-Yao intruder model.
- If a claim is refuted, a report should be presented in the form of a trace. This trace consists out of all actions performed by the intruder, plus the behavior of the model which leads to the disclosure of terms, which have been claimed secret.
- It must be possible to check the model for more than one security hole
- The model-checker should be able to check models which may use symmetrical and asymmetrical encryption.
- It must be possible to have multiple instances of the same role.
- It must be possible to check a model, in which, more than one protocol has been defined. This allows checking situations where the existence of one protocol might influence another.
- Agents must be able to conspire with the intruder. If an agent is a conspirator, all the knowledge that the agent possesses at the start of the model-checking procedure is transferred to the intruder. This allows the intruder to play every trace the agent could have played.
- The model-checker should be built in a modular way. It must be possible to use modules of this model-checker in future implementations.
- There must be a way to specify which roles are initiated and with which variables. This is supported by accepting scenarios.
- The model-checker should have a user interface.

4.4 Limitations

The state space of a model with an unlimited number of runs and an omnipotent attacker is too complex to check. Therefore some limitations must be introduced. Fewer limitations will most likely increase the number of options at each state, broadening the reach of the model-checker, but making it increasingly more costly to compute.

- Only a limited number of runs are considered.
- The model-checker is limited to only work with typed communication; An agent is expecting a message of a certain format and will not accept any other message than a message of the correct format. Because of this approach, type-flaw attacks, in which the intruder tricks an agent to accept a term of the wrong type, will not be detected.
- Only commutative functions have been implemented. So $f(y, x) = f(x, y)$.
- Functions can only produce values of a basic term type.
- Compromised key attacks, in which a key is leaked to the intruder, will not be considered.
- The model-checker will only be provided with a command line interface.

5 SPDL: Security Protocol Description Language

SPDL is the input language for the model-checker. One central idea behind creating SPDL is the ability to check models where multiple protocols are running. Combining this idea with the semantic concept of using role definitions leads to a hierarchy. The top-level consists of the security protcol set, followed by a security protocol. The lowest level consists of role definitions. Input to the model-checking algorithm is divided into two parts. The first part, expressed in SPDL, supplies the protocols to the model-checker. An example of a SPDL script is provided in Section 5.3.1. The second part of the input consists of a scenario which is used to specify the limits of the model. Scenarios are discussed in Section 6.

5.1 SPDL requirements

First and foremost SPDL should allow protocols to be expressed and security claims to be made about those protocols. It should be easy to take an existing protocol and create a definition for it. SPDL should be extendable so other

expression possibilities could be introduced without too much hassle. For example, another kind of claim could be introduced like claim(non-repudiate, term) to express the security property of non-repudiation[3]. SPDL should work intuitively to someone with some experience in the field of security protocols. Roles communicating with each other should be easily defined by copying reads or respectively sends and modifying them. It must be possible to express any protocol which uses symmetrical or asymmetrical encryption and does not require any other term type than agents, nonces, keys, termlists and encterms[4] SDPL supports two types of comments, single line comments start with // and multi line comments start with /* and ends with */.

5.2 Expressing in SPDL

SPDL consists of the following constructs:

- **global** All declarations under global have global scope.
- **function** Used to declare functions.
- **inverse** Used to create an inverse pair.
- **inv** Used to denote an inverse of a function.
- **protocol** Initiates the start of a protocol.
- **endprotocol** Initiates the end of a protocol..
- **var** Used to declare variables.
- **const** Used to declare constants.
- **know** Used to define knowledge sets.
- **def** Initiates start of role actions.
- **role** Used to define roles and implicitly define agents.
- **send** Identifies send operation.
- **read** Identifies read operation.
- **claim** Expresses a claim.

[3]Non-repudiation is a security property which expresses that for some term it can be proven that a certain agent has performed an action on that term.

[4]Some term types, which are not explicitly supported, can still be expressed by combining supported types. For example succ(a) denotes a nonce which is syntactically different from a nonce a. Succ(a) can be used to thwart of replay attacks, enforcing knowledge of a key to create a message containing succ(a) even if a message containing a has been captured. By combining two normal nonces, succ(na) can be expressed by (na,na).

- **secret** Expresses a claim type.
- **agent** Basic term indicating an agent.
- **key** Basic term indicating a key.
- **nonce** Basic term indicating a nonce.

SPDL knows 3 scopes. Global scope, which is the largest. Protocol scope which is smaller than global scope. Role scope which is the smallest scope. Global scope starts at the occurrence of the keyword **global** and runs until the first occurrence of the keyword **protocol**. Protocol scope starts at one occurrence of the keyword **protocol** and runs until the next occurrence of the keyword **endprotocol**. Role scope starts at the first occurrence of the keyword **role** and runs until the next occurrence of the keyword **role**. Role scope is also called local scope. Anything declared within the global scope is accessible by any role of any protocol. Anything declared within the protocol scope is accessible only by roles defined within the scope of that specific protocol. Anything declared at local scope is only accessible within that role. Items declared in smaller scopes overwrite items, of the same name, declared in larger scopes.
There are six types of declarations in SPDL. Functions, Variables, Constants, Knowledge, Inverses, Roles and Definitions. The following subsections will explain these declarations and provide example code to indicate its usage. This example code is not part of any real protocol.

5.2.1 Functions

Functions are used to simulate key types. This way different roles can use different keys, which are of the same type, namely the function, which creates them. In this implementation of SPDL functions are commutative with regard to their parameters. SPDL does not allow zero-ary function definitions. However, the definition of a term typed key as a constant can be used for that purpose. Functions can, thus, have any number of parameters greater than zero. Function parameters can be of any existing type, but SPDL only supports a term of the type key as a value. Functions can be defined after the keyword **function**. An unlimited number of functions can be declared, but they must have unique names. Functions of the same type can be declared by one statement, by separating the function names with commas. Functions declared at smaller scopes overwrite functions declared at larger scopes. The function definition ends when the next keyword is encountered. For example, the declaration of the unary functions PublicKey and SecretKey:

```
Function
  PublicKey,SecretKey: agent -> key
```

5.2.2 Inverses

Inverses are used to express that a certain function has an inverse. If no inverse has been declared, symmetrical encryption is assumed. This means that encryption and decryption must be done by the same key. If two functions have been declared as each other's inverse, then asymmetrical encryption is assumed. This means that a term encrypted with a key, k, can only be decrypted by a key, k', which has been declared to be the inverse of k. A function can have only one inverse. Inverses are declared after the keyword **inverse**. The inverse pairs are comma separated. Inverse definition ends when the next keyword is encountered.
For example, the declaration that the functions PublicKey and SecretKey are each other's inverses:

```
Inverse
  (PublicKey,SecretKey)
```

5.2.3 Variables

Variables refer to terms and are subject to a single-assignment regime. This means that only at one time, during execution,Scan they be assigned a value. This value will then be substituted for every occurrence of the variable. Variables are typed. The only way to declare variables is to specify their type. Each role keeps a symboltable, which keeps track of variables and constants. A variable can only be set at one point in a definition. Once the variable has been set, it becomes a constant. Any variable which has been declared in global scope, will be copied into every role's symboltable. Variables declared in protocol scope will only be copied into the symboltable of roles declared in that same protocol. Variables, declared with role scope, overwrite entries of variables, with the same name, of larger scopes.
A variable declaration starts with the keyword **var** and ends with the next occurrence of a keyword. Multiple variables of the same type can be declared in one statement.
For example the declaration of variable nonces na and nk:

```
var
  nonce: na, nk;
```

5.2.4 Constants

Constants work much like variables. The only difference is that they have a value from instantiation. They are kept, like variables, in the symboltable of a role and have the same scoping rules as variables. Constants do not represent a specific datatype, but rather a definition of the type. This allows for fully symbolic treatment of constants. For example, the declaration of constant nonces nb and nl:

```
const
  nonce: nb, nl;
```

5.2.5 Role

A role can be seen as the skeletal of a run. Every basic term, a run will ever use in its life, must be declared. The only information a role still needs before it can be instantiated, will be provided to it, by its parameters. All parameters of a role can be seen as variables of the type agent. Every role has at least one parameter. That parameter defines which agent controls the role. The rest of the role parameters define with which agents it can communicate. A role definition starts with the keyword role, followed by the controlling agent and the communicating agents. A role declaration can be divided into two parts. The first part defines the terms which will be used by the second part of the role declaration. The second part defines the behavior of the role.
For example, a role which is controlled by agent A and communicates with agent B. (The definition part is explained further below in Section 5.2.7)

```
role A(B);
    var
      nb: nonce;
    const
      na: nonce;
```

5.2.6 Knowledge

If a role uses a function, or a value of a function, which has been declared in protocol, or in global scope, that function or value must be declared known. This allows the model-checker to be aware of all the knowledge a run has before the protocol executes. If a run is controlled by an agent who conspires with the intruder, all its knowledge is transferred to the intruder. Knowledge declaration starts with the keyword **know** and ends with the next occurrence of a keyword. Knowledge can only be declared at role scope.
For example, a role, controlled by agent A, which knows the public key of every agent and as well as its own secret key.

```
know
  PublicKey, SecretKey(A);
```

5.2.7 Definitions

The definition is the most important part of a role. It specifies the behavior, which is the sequence in which actions occur. Within a definition more complex terms can be constructed out of previously declared variables and

constants. These newly created terms do not have entries in the symbol-table. As previously explained there are three actions which can occur in a definition: sends, reads and claims. Every action starts by specifying its type. Sends and reads must include the intended destination and the expected origin, respectively. This information is needed in order to present a comprehensible trace, if any security hole is found. Every action contains one and only one term. If in one message multiple terms must be sent, then a term of the type termlist must be used.
Send and claim actions may only be performed on terms which contain no variables. Reads can be performed on a term which may include variables. Such a term is called a context. A role, executing a read action, will only accept a term, if all the constants in the context are exact matches with the constants in the incoming term and all the variables in the context are of the same type as the constants in the incoming term. It is said that a term fills the context.
A definition can only be declared at role scope and starts with the keyword **def** and ends with the next keyword occurrence.
For example, a role controlled by A, communicating with B having all the above declared elements might have the following sample definition:

```
def
  send(B, {(na)}PublicKey(B));
  read(B, {(na,nb)}PublicKey(A));
  send(B, {(nb)}SecretKey(A));
  claim(secret,(na));
```

With asymmetrical encryption, agent A can encrypt terms with its secret key. This creates a form of authentication, since any agent who knows agent A's public key, can be sure that the term was encrypted by no other agent then agent A. However, if agent B would have an action reading this term like:

```
  read(A, {term}SecretKey(A));
```

It would suggest that agent B knows the secret key of agent A. This of course would not be the design of the protocol, since secret keys are in fact secret. Therefore a new expression had to be introduced. Since B would know the public key of A, it would suffice to express that B is reading a message which has been encrypted with the inverse of the public key of A.

```
  read(A, {term}inv(PublicKey(A));
```

This way a compromised agent B would not be able to provide the intruder with A's secret key.

5.3 The grammer of SPDL

In the grammar of SPDL the following notation is used:
`(text)*` means text can be repeated 0 or more times.
`[text]` means text can be repeated 0 or 1 time. (optional).
`|` denotes a choice.
Non-terminals are surounded by `<` and `>`, while terminals are surounded by `"`.

```
<program>               ::= <global_declaration>
                              (<protocol_declaration>)*
<global_declaration>    ::= global <declaration>
<declaration>           ::= [<function_declaration>]
                              [<var_declaration>]
                              [<const_declaration>]
                              [<inverse_declaration>]
                              [<know_declaration>]
<function_declaration>  ::= function (<function_definition>)*
<function_definition>   ::= <identifier> (","<identifier>)*":"
                              <term>
                              ("," <term>)* "->"
                              <term>("," <term>)*";"
<var_declaration>       ::= "var" (<var_definition>)*
<var_definition>        ::= <identifier>
                              (","<identifier>)*":"<term_type>;
<const_declaration>     ::= "const"
                              (<const_definition>)*
<const_definition>      ::= <identifier>
                              (","<identifier>)*":"<term_type>;
<know_declaration>      ::= "know" <identifier> (","<identifier>)*";"
<inverse_declaration>   ::= "inv" (<identifier>";"|
                                      <identifier>"("
                                        <identifier>(","<identifier>)*
                                      ");"
                                     )*
<protocol_declaration>  ::= "protocol" <identifier>";"
                               <protocol_definition>";"
                               "endprotocol;"
<protocol_definition>   ::= <declaration>
                               (<role_declaration>)*
<role_declaration>      ::= "role"
                               <identifier>([<identifier>
                               (","<identifier>)*])";"
                               <declaration>
                               [<def_declaration>]
```

```
<def_declaration>       ::= "def" (<sequential_statement>";")*
<sequential_statement>  ::= "send"(<identifier>,<term>)
                          "read"(<identifier>","<term>)
                          "claim"(<claim_type>","<term>)
<claim_type>            ::= "secret"
<term_type>             ::= "agent"|"nonce"|"key"
<term>                  ::= <identifier>          |
                          "("(<term>)")*"<term>   |
                          <identifier>(<term>)    |
                          (<term>(","<term>)*)
```

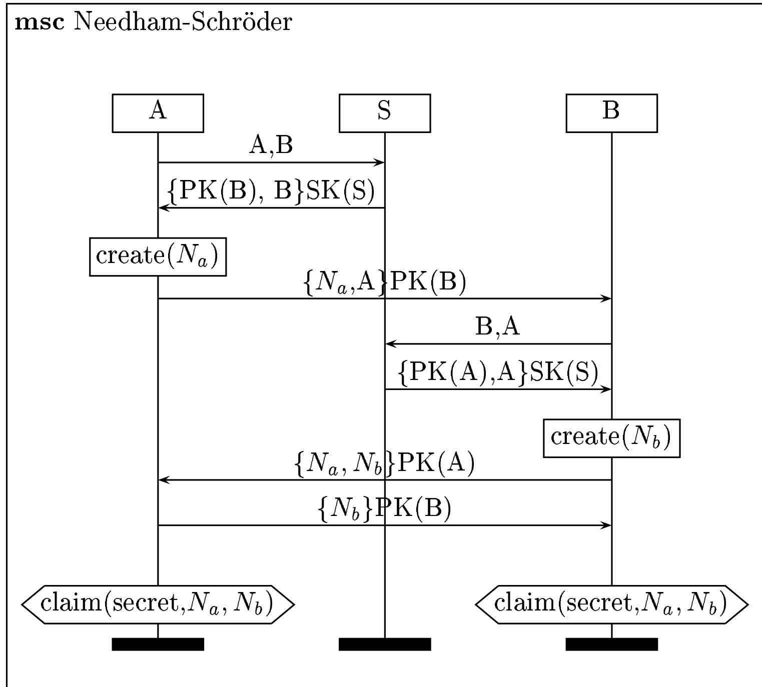
Apart from checking the structure of the input file to the rules expressed by the BNF, the interpreter also checks the input to be composed of semantically correct expressions. Semantically correct input adheres to the following rules.

- There can be only one declaration of some symbol name in any scope.
- Only functions can be expressed as each others inverses.
- Arguments of functions declared in the **know** definition must be constants.
- Any function used in the **def** clause, must be declared in the **know** clause in the same scope.
- Send and Claim actions can only be executed on terms which do not contain variables.
- The expression **inv** can only be called on keys, which have an inverse.

5.3.1 Needham-Schroeder public-key protocol SPDL example

The SPDL input file for the famous Needham-Schroeder [7] public-key protocol is presented as an example of SPDL. The Needham-Schroeder public-key protocol is a protocol which ensures two agents to be communicating with each other and sharing secrets. The protocol uses a server to exchange public keys. Needham-Schroeder has been used for a decade, before Gavin Lowe proved it to contain a security flaw and corrected it in the Needham-Schroeder-Lowe [6] public-key protocol.

The Needham-Schroeder public-key protocol can best be explained by a MSC (Message Sequence Chart).



This protocol can be expressed in SPDL as follows:

```
global
protocol NSPK;
  function
    KS: agent -> key;
    KP: agent -> key;
  inverse
    (KS,KP);
  role A(B,S);
    var
      nb: nonce;
      KPb: key;
    const
      na: nonce;
    know
      KS(A); KP(A); KP(S);
    def
      send(S,(A,B));
      read(S,{(KPb,B)}inv(KP(S)));
      send(B,{(na,A)}KPb);
      read(B,{(na, nb)}inv(KS(A)));
      send(B,{nb}KPb);
      claim(secret,(nb,na));
  role S(A,B);
    know
      KP; KS(S);
```

```
    def
      read(A,(A,B));
      send(A,{(KP(B),B)}KS(S));
      read(B,(B,A));
      send(B,{(KP(A),A)}KS(S));
  role B(A,S);
    var
      na: nonce;
      KPa: key;
    const
      nb: nonce;
    know
      KS(B); KP(B); KP(S);
    def
      read(A,{(na,A)}inv(KS(B)));
      send(S,(B,A));
      read(S,{(KPa, A)}inv(KP(S)));
      send(A,{(na, nb)}KPa);
      claim(secret,(nb,na));
endprotocol;
```

6 Scenarios

As explained before in Section 5, input to the model-checker has been divided into two parts. The first part, expressed in SPDL, has been explained in the previous section. The second part, scenarios, will be explained here.
One of the limitations imposed on the model is the number of runs. In the scenario the state space is bounded by the number of defined runs. Without the limitation of the number of runs every possible scenario would have to be checked. In scenarios two types of runs can be defined. There can be runs, which are controlled by the intruder and runs which are controlled by trusted agents. A trusted agent, or honest agent, does not intend to help the intruder. Trusted agents by definition follow their role description precisely and do not perform any other action than this role description. Trusted agents will therefore never leak information in any other way than defined by their role description.
In the semantics for security protocols, there is no instantiation of the intruder. In order to still specify roles which are controlled by the intruder, untrusted agents are introduced.
Untrusted agents are agents that conspire with the intruder. A run controlled by an untrusted agent is considered compromised. Compromised runs will not be added to the model in the same way normal runs are. A compromised run is added to the model by extracting all knowledge of that run and adding it to the initial intruder knowledge. The intruder can then create every message defined by that runs role.
Because protocols are being defined in a separate SPDL file, it is easy to try out many scenarios without having to rewrite the protocol. Furthermore attacks on one protocol can be easily tried on another protocol by using the same scenario.
In order to be able to express scenarios, a simple scenario language has been created. A scenario expressed by this scenario language consists of 2 parts; an agent part where all agents are declared and a run part where runs are instantiated. Agents

can be declared to be trusted and untrusted. Instantiating a role with an untrusted agent running the role will create a conspirator. Conspirators are not included in the set of runs on which the algorithm will be executed, but rather their knowledge is dumped in the $K_{intruder}$ so the intruder can "play" all their actions. Trusted agents controlling a run, with an untrusted communication parameter, will have all their claims removed. Roles can be instantiated with wildcards "*". This will cause runs to be instantiated where the wildcard will be filled with every agent defined. The scenario language supports comments just like SPDL does.

6.1 The BNF of scenario language

```
<scenario>          := scenario
                     <agent_dec>
                     [<run_dec>]
<agent_dec>         := "agent" [<agent_def>]
<agent_def>         := (trusted | untrusted)":" <id>(","<id>)*";"
<run_dec>           := run (run_def)*
<run_def>           := <id>"."<id>":"<id>|"*" "(" ["*"|<id> (",*"|<id>)"*)";
```

Scenarios must adhere to some basic rules to ensure semantical correctness.

- In an agent declaration each symbol representing an agent can only be used once.
- The protocol and the role value in a run declaration must have been declared in the SPDL input file.
- Agents in a run declaration must have been declared in the agent declaration.

6.1.1 Needham-Schroeder public key scenario example

A scenario for the Needham-Schroeder protocol is presented as a running example of a scenario input file. By running this scenario with a of definition Needham-Schroeder in SPDL, the security holes found by Lowe in 1995 can be reconstructed. This scenario file defines three trusted agents, A, B and S as well as an untrusted M. It then instantiates six runs, of which two will be played by the intruder.

```
scenario

agent
trusted: A,B,S;
untrusted: M;

run
NSPK.A:A(B,S);
NSPK.B:B(A,S);
NSPK.S:S(A,B);
NSPK.S:S(A,M);
NSPK.A:M(B,S);
NSPK.A:M(M,S);
```

7 Design & implementation

The system was designed bottom up and completely written in the Java 1.4 language. By working out examples on paper an abstract algorithm could be designed. From this abstract algorithm, every aspect was carefully implemented.
This section starts out by giving an example of how the secrecy security goal could be verified on paper. The algorithm used by the model-checker is discussed next, followed by a discussion of the modules this program is composed of. It ends by discussing how the initialization process uses the user input to initialize the system.

7.1 The algorithm

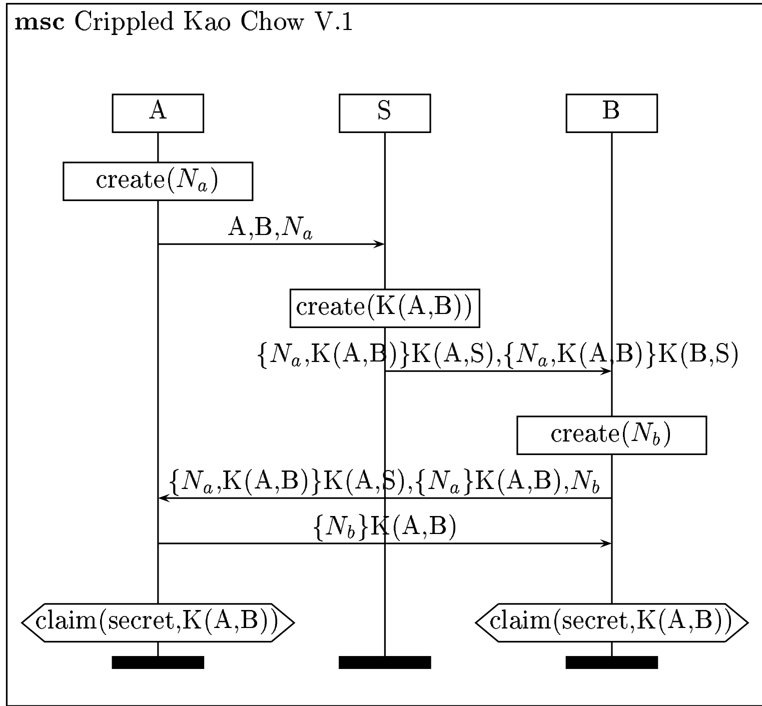
The algorithm checks the model in a brute force manner. However, since a full brute force check of the model will yield exponential time and space problems, this model-checker only checks a subset of all possible traces. This subset is limited by the scenario and by the use of supertransitions instead of transitions.
For exactly those events which can execute in any order, order can be abstracted by compiling them in a set. From this set supertransitions can be created in which ordering has no effect[5]. All possible traces a protocol can perform can be seen as a tree. The algorithm will follow each path in the tree. Each time a node is encountered, it will validate the set of claims by checking the contents of the intruder knowledge to the terms claimed secret. If at some node, one or more claims can be refuted, it will print the trace from the root till the conflicting node.
The best way to show how this algorithm works is by example.

7.1.1 Crippled Kao Chow Version 1

The Kao Chow version 1 protocol [4] by I Long Kao and Randy Chow (1995) is an authentication protocol which uses a server, S, to distribute symmetric keys to the agents A and B. After roles A,B and S have completed their runs, it claims that the symmetric key K_{ab}, which A and B use to communicate, has not and will never be in the intruder knowledge. Kao Chow Version 1 contains a security hole that cannot be detected by our current `checkSecrecy` algorithm[6] (Section 7.1.2). In order to show how a hole can be found, authentication information is removed from the protocol. The resulting protocol, Crippled Kao Chow Version 1 (CKCV1) can be expressed by the following MSC:

[5]Correctness of this algorithm is not the object of this graduation thesis.

[6]The known attack of this protocol is a compromised key attack, which this version of the model-checker cannot detect



A SPDL file expressing CKCV1 could be constructed as follows:

```
global
function
  K: agent,agent -> key;
protocol KCV1;
  role Initiator(Responder,Server);
    var
      nb: nonce;
      kas,kab: key;
    const
      na: nonce;
    know
      K(kas);
    def
      send(Server, (Initiator,Responder,na));
      read(Responder, ({(na,kab)}K(A,S),{na}kab,nb));
      send(Responder, {nb}kab);
      claim(secret,kab);
  role Server(Initiator,Responder);
    var
      na,nb: nonce;
    know
      K(A,S),K(A,B),K(B,S);
    def
      read(Initiator, (Initiator,Responder,na));
      send(Responder, ({(na,K(A,B))}K(A,S),{(na,K(A,B))}K(B,S)));
  role Responder(Initiator, Server);
```

```
    var
      na: nonce;
      kas,kab:key;
    const
      nb: nonce;
    know
      K(B,S);
    def
      read(Server, ({(na,kab)}kas,{(na,kab)}K(B,S)));
      send(Initiator, ({(na,kab)}kas,{na}kab,nb));
      read(Initiator, {nb}kab);
      claim(secret,kab);
endprotocol;
```

This example discusses a walkthrough of the algorithm using the following scenario:

```
scenario
agent
trusted: A, B, S;
untrusted: M;

run
CKV1.Initiator:A(B,S); //assigned runId 1 (automatically)
CKV1.Responder:M(A,S); //assigned runId 2 (automatically)
CKV1.Server:S(A,B);    //assigned runId 3 (automatically)
```

Because M is an untrusted agent all knowledge in Run CKV1.Responder:M(A,S), which consists of K(M,S) and nb#1 is transferred to the intruder. The initial intruder knowledge therefore consists of $[n_i, k_i, A, B, S, M, K(M,S), n_b\#2]$.
Following the definitions in the scenario, the following two runs can be instantiated. The run controlled by the compromised agent M, will not be instantiated. It will be played by the intruder.

Action	Initiator:A(B,S)
1	send(S,$(A, B, N_a\#1)$)
2	read(B,($\{N_a\#1, kab\}K(A,S)$,$\{N_a\#1\}kab$,N_b)
3	send(B,$\{N_b\}kab$)
4	claim(secret,kab)

Action	Server:S(A,M)
1	read(A, (A,B,N_a))
2	send(B, $(N_a, K(A,M)K(A,S), N_a\#1, K(A,M)K(M,S))$)

After initialization all runs will execute their first actions until a read is encountered, or no actions are left. The state after initialization is the root state. The only action executed in this stage for this scenario is Initiator:A(B,S)'s first action; $A \rightarrow S : (A, B, N_a\#1)$

State: Root

$K_{intruder} = [n_i, k_i, A, B, S, M, K(M,S), n_b\#1] \cup [N_a\#1]$

This state has the following structure:

Action	Initiator:A(B,S)	Server:S(A.B)
1	read(B,($\{N_a\#1, kab\}K(A,S)$)	read(A, (A,M,N_a))

Since the intruder knowledge does not contain K(A,S) or a term which has been encoded with the key K(A,S), it cannot inject a message which will be accepted Initiator:A(B,S). However the intruder has the following possibilities for Server:S(A,B).

Possible actions:

I A → S: $(A, M, N_a\#1)$
II A → S: $(A, M, N_b\#2)$
III A → S: (A, M, N_i)

Actions I,II and III can be combined to form the following transitions:

I II III

Each transition will lead to a new state.

State: I

After executing action I, Server:S(A,B) performs the action
S → B: $\{N_a\#1, K(A,M)\}K(A,S), \{N_a\#1, K(A,M)\}K(B,S)$ and the intruder knowledge is increased to:
$K_{intruder} = [n_i, k_i, A, B, S, M, K(M,S), n_b\#1, N_a\#1] \cup$
$[\{N_a\#1, K(A,M)\}K(A,S), \{N_a\#1, K(A,M)\}K(M,S)]$, but
because K(M,S) $\in K_{intruder}$
$K_{intruder}$ becomes: $[n_i, k_i, A, B, S, M, K(M,S), n_b\#1, N_a\#1] \cup$
$[\{N_a\#1, K(A,B)\}K(A,S), K(A,M)]$

This state has the following structure:

Action	Initiator:A(B,S)	Server:S(A.B)
1	read(B,($\{N_a\#1, kab\}K(A,S)$,$\{N_a\#1\}kab$,N_b)	no more actions

The intruder cannot alter the key kab in the term $\{N_a\#1, kab\}K(A,S)$, because it doesn't know K(A,S) and there are no other terms matching $\{N_a\#1, kab\}K(A,S)$ in the intruder knowledge. Since the intruder knows the key kab, which value is K(A,M), it can use it to create the term $\{N_a\#1\}kab$. For N_b the intruder has three options. Combined this leads to Possible actions:

IV : S → A: ($\{N_a\#1, kab\}K(A,S)$,$\{N_a\#1\}kab$,$N_a\#1$
V : S → A: ($\{N_a\#1, kab\}K(A,S)$,$\{N_a\#1\}kab$,N_i
VI : S → A: ($\{N_a\#1, kab\}K(A,S)$,$\{N_a\#1\}kab$,$N_b\#2$

IV

After accepting the injection, Initiator:A(B,S) performs, in sequence,
A → B: $\{N_b\}kab$ and claim(secret, kab).
The action which A → B: $\{N_b\}kab$ does not alter the intruder knowledge.
The claim, however, does not hold. The value of kab is K(A,M) and this value is in the intruder knowledge, thus, this claim is refuted. The trace breaking the protocol is:

send	[Initiator:A(B,S)@1]	$A \to S : (A, B, N_a\#1)$
inject	[Server:S(A.B)@3]	$A \to S : (A, B, N_a\#1)$
send	[Server:S(A.B)@3]	$S \to B : \{N_a, K(A, M)\}K(A, S)$, $\{N_a\#1, K(A, M)\}K(M, S)$
inject	[Initiator:A(B,S)@1]	$B \to A : \{N_a, K(A, M)\}K(A, S)$, $\{N_a\#1, K(A, M)\}K(M, S), N_a\#1$
send	[Initiator:A(B,S)@1]	$A \to B : \{N_a\#1\}K(A, M)$
claim	[Initiator:A(B,S)@1]	$A : SECRETK(A, M)$

Allthough a breach of the protocol has been found, other breaches might be found by backtracking to the last state which does not breach the protocol. This is the root state. From the root states II and III can be explored, but will not produce new breaches.

7.1.2 The checkSecrecy algorithm

The example in Section 7.1.1 illustrates how the algorithm examines each state, finds possible injections and checks the effect of these injections. The problem becomes more complex if more runs would be included. More runs might produce states where multiple injections are possible on multiple runs. The first job of the function `checkSecrecy` is to find all supertransitions. These supertransitions consist of injection sets that indicate which injection will be used, on which run, and which injection will not be used. Supertransitions consist of unique injections sets in which the order in which these injections are executed is irrelevant. Each supertransition is checked recursively. The algorithm uses a depth first approach to check states.
The `checkSecrecy` algorithm is defined in the ConfCheck module. Stripped down from debug information and flag implementation the main algorithm, `checkSecrecy`, could be defined as:

```
TermSet checkSecrecy(RunSet R, IllegalInjections I,
Knowledge K, TermSet C){

  TermSet failedClaims;
  /* K, C and possibly R are modified*/
  advanceToRead(R,K,C);
  failedClaims = checkClaims(C,K);
  if(isEmpty(S)){
    if(isEmpty(failedClaims)){
      return null;
    }else{
      return failedClaims;
    }
  }
  StepSet S = superTransitions(R,I,K);
  if(empty(S)){
```

```
    return null;
  }
  foreach(step s in S){
   /* returns new updateds instance of R */
    Run newR = updateRun(R,s);
    IllegalInjections newI = expand(I,illegal(s));
    Knowledge newK = clone(K);
    TermSet newC = clone(C);
    failedClaims = checkSecrecy(newR,newI,newK,newC);
    if(!isEmpty(failedClaims)){
      return failedClaims;
    }
  }
}
```

The parameters of the recursive function `checkSecrecy` have been explained in previous sections. In this algorithm, the function `advanceToRead`, takes the set of Runs and executes, for each Run, actions until a read action is encountered or no actions are left. Send actions in a Run can increase the intruder knowledge[7] and claim actions alter the set of claims. Therefore Knowledge en the set of claims is provided to this function as arguments.
After execution of the function `advanceToRead` the model-checker finds itself in a new state. If from this state the algorithm can reach other states, the state is a node. The state is a leaf if there is no other, unvisited, state to be reached. If a leaf is reached, the algorithm backtracks to the nearest node, which still has unvisited states. Backtracking has no side effects to the parameters of the `checkSecrecy` algorithm. This is insured by using clones[8] in any recursive call.
In each state claims might have become refutable as a side effect of the function `advanceToRead`. The function `checkClaims`, checks the intruder knowledge for any occurrence of the terms in the set of claims. If indeed one or more claims can be refuted, the algorithm backtracks, by returning the failed claims.
If no claim failed, the algorithm determines every possible transition[9] This is done by calling the function `superTransitions`. This function needs three arguments; The set of Runs which is used to determine which terms are expected by which Runs. The intruder knowledge which it needs to figure out which expected terms it could create. Since some Injections might be illegal in this state, `superTransitions` also needs to receive the IllegalInjections datastructure as an argument. The function `superTransitions` first retrieves the set of all possible Injections. This part of the function is handled by another function called `getAllPossibleInjections`.(Section 7.1.4)
When all possible injections have been retrieved combinations are made. Steps are then created by combining each Injection for each Run with any number of Injections of other Runs. Each Injection which is not is not performed is then added

[7] The semantics for security protocols define the network in total control of the intruder. A send action delivers the term to the intruder who decides what to do with it.

[8] The clones sent are a combination of deep and shallow-clones. Terms of course will never be cloned after initialisation. Because of this TermReferences can be cloned easily.

[9] A transition is a possible Step, which can be made from some state. As mentioned in previous sections Steps also include illegal Injections.

to the Step's IllegalInjections datastructure. Each Step found by this process is then performed by updating the parameters of the function `checkSecrecy`, and recursively calling the function with these new parameters until all Steps have been performed or a claim has been refuted.

7.1.3 The getAllPossibleInjections algorithm

The stripped algorithm of the function `getAllPossibleInjections` can be defined as:

```
InjectionSet getAllPossibleInjections(RunSet R,
  IllegalInjections I, Knowledge K){

  InjectionSet result = new InjectionSet();
  foreach(run in R){
    if(run.status != COMPLETED){
      Term context = run.getCurrentContext();
      TermSet ts = K.fillContext(context);
      InjectionSet options = convertToInjections(ts,run.id);
      result.addAll(remIllegalInjections(options, I));
    }
  }
  return result;
}
```

The `getAllPossibleInjections` algorithm retrieves for every Run, which has not been completed the context. Runs support this method by retrieving the target[10] TermReference from the current read action. This TermReference is the recursively visited[11]. At each recursion level, the TermReference encountered is converted to the Term it refers to. The EncTermRef and TermListRef types will cause EncTerm and respectively TermList types to be created. For each set[12] BasicTermRef encountered within the target TermReference, the Term it references to is copied to the newly formed context Term. If an unset BasicTermRef is encountered a special term of the type *hole* is created. An instantiated Hole keeps track of the term type it represents and the name[13] of the term it represents. A Hole should be seen as a "hole" in the context. If a term t should match a context c, every basic term in the c should be exactly the same term as the its respective counterpart in t. Every Hole in c, should be matched by a term in t, which is of the same type as the hole type. The term *hole* performs a kind of prototyping in a context.
After a Run delivers a context, all terms which can be constructed with the current intruder knowledge should be acquired. This is handled by a method of Knowledge called `fillContext`.

[10]The target TermReference is the TermReference in a read Action, which models the context of the read action.

[11]This TermReference can be of any type and may contain any level of nesting.

[12]A set BasicTermRef is a constant and contains a reference to an instantiated term.

[13]The internal name of the Hole matches that of the variable specified in the SPDL file.

7.1.4 The fillContext algorithm

This method, as explained before, is the most fundamental function of the application. It was also the most complicated one to construct. It analyzes the context presented in its parameter and retrieves all possible terms from the context which match the context. The basic idea behind the algorithm behind `fillContext` is:

- Get a list of all holes in the context.
- Find all required substitutions[14]. Required substitutions have special properties, this will be explained shortly.
- Then for each remaining hole, find options for that hole and use this information together with the required substations to create mappings of holes to terms.
- Finally create terms matching the context by using the context as a prototype and the mappings as roadmaps.

The `fillContext` algorithm is based on the unification. The semantics for security protocols in development, currently, only defines unify for untyped communication. However, the general idea behind this concept for typed communication is the same. Unify is a function which is used by the intruder to create a term based on a context. The context is provided by an agent who is trying to read a message. An agent who is trying to read a term defined by the context, c, will accept a term, t, if t unifies c.

The following algorithm is a simplified version of the implemented algorithm of `FillContext`.

```
TermSet fillcontext(Term context){
  Set holeDistr, holes, result,
  required, optional,

  getHoles(context, holes, required, optional);

  if(empty(holes)){
    return Fillcontext(context,null);
  }

  if(empty(required)){
    holeDistr.add(getHoleDistr(context,holes,null));
  }else{
    foreach(predef in required){
      holeDistr.add(getHoleDistr(context,holes,predef));
    }
  }

  holeDistr = combine(holeDistr, optional);

  foreach(HST in holeDistr){
```

[14]A substitution in this case is the substitution of a hole by some term.

```
    Set termSet = fillcontext(context,HST);
    results.addAll(termSet);
  }
  return termSet;
}
```

The function `getHoles` analyzes a context. Its parameter, holes, is filled with an index of all Holes encountered in the context.
The parameter required is filled with mappings of holes to terms. These mappings consist of substitutions which the intruder is forced to make. EncTerms in the context, which are encrypted with keys, that can not be derived[15] from the intruder knowledge, can only be substituted by EncTerms, encrypted by the same key, which are available in the intruder knowledge. There might be several options for this substitution, but any choice will force a substitution on the context with terms which might not be derivable in the intruder knowledge. Since the contents of EncTerms with underivable keys are not in the intruder knowledge, they wouldn't be considered to match a hole by any subsequent steps in the algorithm.
The optional parameter is filled with other hole mappings to terms. These mappings represent mappings of EncTerms with derivable keys. The difference between these mappings and the required set of mappings is that the required mappings leave the intruder with no choice. These optional mappings may be pre-empted by required mappings later in the algorithm.
If a term contains no holes, an unifiable term is sought with no mappings by the call `fillContext(context,null)`. This is a secondary function, which fills a context when all mappings for all holes are known. If there are no required mappings, the function `getHoleDistr`, creates mappings for EncTerms in the context which do not contain holes in the contents, but have a derivable key. All these mappings are then combined, with mappings from the required set overwriting those in the optional set.

Take for example the context $c = (\{*_{n_b}\}k_1, \{*_{n_b})\}k_2)$ and the intruder knowledge, $K_{intruder}$, consisting of n_a, n_b, k_1, $\{n_b\}k_2$,$\{n_c\}k_2$ where $*_{n_b}$ is a hole of the type nonce with the name n_b, n_a,n_c are a nonces and k_1,K_2 are keys. Since k_1,n_a and $n_b \in K_{intruder}$ the intruder can form $\{n_b\}k_1$ and $\{n_a\}k_1$ to match the subcontext $\{*_{n_b}\}k_1$. However for subcontext $\{*_{n_b})\}k_2$ the intruder has also has two choices, namely $\{n_b\}k_2$ and $\{n_c\}k_2$. Because $k_2 \notin K_{intruder}$ these two options appear as separate mappings in the required set. The first option, $\{n_b\}k_2$, pre-empts $\{n_a\}k_1$, leaving a mapping of $*_{n_b} \rightarrow n_b$. The second required option pre-empts both $\{n_b\}k_1$ and $\{n_a\}k_1$, leaving no suitable mapping. With this intruder knowledge, the only term which unifies this context is $t = (\{n_b\}k_1, \{n_b)\}k_2)$.

7.2 Modules

The model-checker consists of four main modules, which all have been implemented and placed in their own directory. These modules are:

[15] A term can be derived from the intruder knowledge, if it can be constructed from terms in the intruder knowledge.

- Knowledge
  Knowledge can be seen as the core of the application. It has no dependencies to any of the other modules, while all other parts depend on it. It includes all data structures and algorithms to create a class which can serve as the intruder knowledge as well as the definition of terms.

- Role and Runs
  This module contains all data structures and algorithms to create Roles and Runs. Roles will only be created at SPDL parse time. Roles are used at scenario parse time to create Runs. Runs are implemented as a stripped down versions of Roles. All administration structures needed at parse time are removed and functionality is added in order to be able to perform needed by the model-checker.

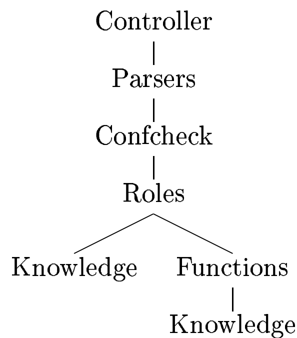
- Functions
  This module contains all the data structures and algorithms to enable SPDL to use functions. It improves code modularity, if every main SPDL concept could be implemented as a separate module. Therefore functions have not been defined within the Role module, but in a separate module.

- Parser
  This module contains the parser. Since two separate and different input languages have been created, there are also two parsers. These two parsers are used by the controlling process to appear as one entity.

- ConfidentialityChecker
  The module ConfidentialityChecker includes all data structures and algorithms needed to check the model for secrecy. Separating the confidentiality checker as a module also improves code modularity. Model-checkers for other security properties can be plugged in, without disturbing the confidentiality checker.

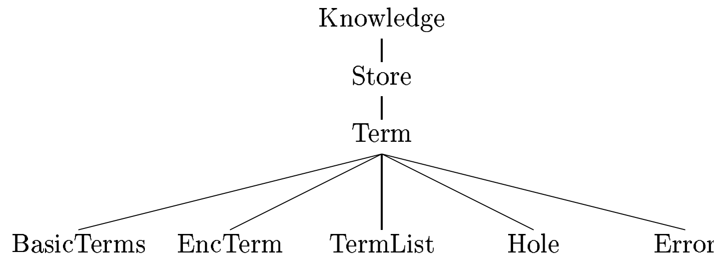
- Controller
  The controller forms the interface, which is used by the end user of the system. This interface allows the specification of input files, as well as the toggling of specific flags. The controller makes sure that all the required initialization steps are executed in the right order, then checks the model using the Confidentiality checker and finally returns control to the end user.

7.2.1 Dependency Tree



7.2.2 Knowledge

Knowledge has been structured in the following way:



At the basis of the intruder knowledge are terms. The class Term is the superclass for all term types. There are currently five term types, which can be used in SPDL scripts; *nonce*, *key*[16], *agent*, *termlist* and *encterm*. There are two internal types; *hole*, which is used to specify a variable in a context and *error*, which is not used currently. The class Term contains integer type, which denotes the term type it represents. Integer constants are used to express the type of a term.
The basic types, agent, nonce and key are implemented as classes with the class Term as their superclass. By initiating the superclass with an appropriate integer representing its type, instances of one of the basic term types, can still be checked for their type, even if they are cast down to the class Term.
The class SPDLKey provides a method by which an inverse can be supplied. If a key has been supplied an inverse, asymmetrical encryption is assumed.
EncTerm, which expresses a term encrypted by another, is implemented as a class EncTerm. Encterms accepts two terms, an encrypted term and the key which encrypted it.
TermList, which expresses a composed term, is implemented as the class TermList. TermLists can hold all terms in the composition. After a TermList object is instantiated, terms are added to it. TermLists keep administration of their contents in two ways. Every term added to it is kept in a Vector. Vectors are by definition ordered and so the order in which terms are added must be the order in which they appear in the termlist. The second part of the administration is keeping track of encrypted items within the list. A TermList can therefore easily tell what items in its contents are encrypted with what key. This is necessary because termlists can contain termlists and so on. It would be a very costly operation when a key is found to find out what terms can be decrypted.
The next implementation level of intruder knowledge is the Store. The Store is a container which can hold terms. It is implemented by the class Store. A Store is in someway similar to a termlist. Like a TermList it can hold terms and like a TermList it keeps administration about which keys are used to encrypt which terms. However a Store is not a Term and terms Stored in the Store are not ordered. The Store itself cannot be used as intruder knowledge. The intruder knowledge must be closed and the Store does not ensure closure with regard to the operations encryption & decryption and composition & decomposition.
Intruder knowledge is implemented as an interface for the Store, which ensures the closure property. This interface is implemented by the class Knowledge. The most

[16]Keys are implemented by the class SPDLKey, in order not to pollute the name space. (The class java.util.Key also exists.).

fundamental and important operation is the method `fillContext` that is provided by the class Knowledge. The method `fillContext` accepts a context[17] as an argument and retrieves the complete set of terms, from the intruder knowledge, which match the context. This means that from all elements in the intruder knowledge compositions and encryptions will be constructed in an attempt to find matching terms. This function is fundamental, because each of these matching terms will be used to attack the protocol. If any term could match but is not retrieved at this point, the model-checker will miss that attack and possibly miss a security hole. The `fillContext` algorithm will be discussed further in this document. Following is a schematic description of the most important methods of these datastructures.

Term			
Name	Return	Parameters	Action
IsBasicTerm-Type	Boolean	None	Returns true if term is a basic type.

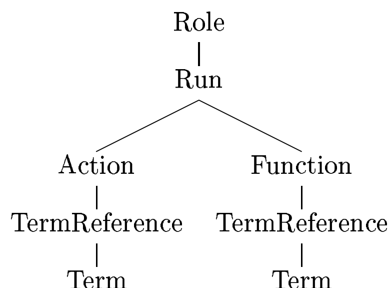
Store			
Name	Return	Parameters	Action
Insert	none	Term T	Inserts T into Store.
Contains	Boolean	Term T	Returns true if the Store contains T.
TermsEnco-dedWith	Set	Term T	Returns the set of all terms encoded with T.

Knowledge			
Name	Return	Parameters	Action
Insert	none	Term T	Inserts T in the knowledge and closes it.
Derivable	Boolean	Term T	Returns true if T exists in the Knowledge or can be constructed from terms in the Knowledge.
Unifies	Boolean	Term T, Context C	Returns true if T unifies C.
Unifiable	Boolean	Context C	Returns true if a term T can be created that unifies T with C.
FillContext	Set	Context C	Returns a Set of all terms in the knowledge which unify C.

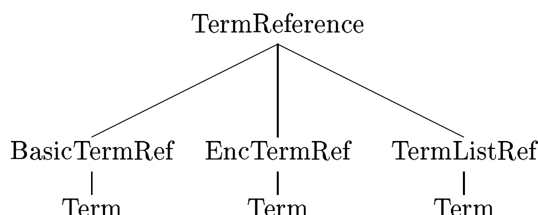
7.2.3 Roles and Runs

Roles and Runs have been structured in the following way:

[17] A context is a term that may contain variables.



As explained before, a Role can be seen as an uninstantiated Run. A Role is build from a specification of a role out of a SPDL file. It keeps administration of variables and constants used by its definition as well as functions. Furthermore it must keep track of an ordered sequence of actions.
Roles keep track of variables and constants by the use of TermReferences and a symboltable. TermReferences are special data structures which can point to terms. There are three kinds of TermReferences, which are implemented much the same as terms. A super class, TermReference, has been constructed and every kind of TermReference must implement it. Like terms, instances of TermReferences use integer constant to indicate its kind. TermReferences have been structured in the following way:



The class BasicTermRef holds pointers to basic terms. BasicTermRefs can only point to one type of basic term. When creating a BasicTermRef that type, in the form of an integer constant, must be supplied and will not change in the entire lifespan of that BasicTermRef. If a BasicTermRef doesn't point to a term, it can be seen as a variable. It is said that a BasicTermRef has not been set. A BasicTermRef can only be set once, by supplying it the term to which it will point. Once it points to a term, it can be seen as a constant.
The class EncTermRef holds reference to Encterms. However, instead of pointing to an EncTerm object, it contains TermReferences which only point to the contents of the EncTerm. These are a term which has been deemed encrypted and a term which is said to encrypt that term.
The class TermListRef keeps an internal Vector of TermReferences.
Because TermReferences can be any of the three TermList types, EncTermRef and TermListRef can be used as pointers to any EncTerm and TermList respectively.
The symboltable only holds references to BasicTermRefs. It is implemented by a hashtable, which keeps the name of the term as it appears in the var or const definition of the SPDL input file. At parse time all terms declared within the const definition will have an entry in the symboltable which points to a set BasicTermRef. Constants are thus created at parse time. Variables declared in the var definition will be represented by an unset BasicTermRef entry in the symboltable.
Actions are specified in the def definition in a SPDL input file. There have been four

action types implemented. Three of them can be used in SPDL specification and one is used internally. Actions are implemented by the class Action and are used by Roles and Runs. Actions keep track of their type, which is an integer constant. The type of an Action is specified at parse time and cannot be changed once the Action has been instantiated. The three user Actions are of course, send, read, and claim. These actions are always created by instantiating an action with the respective type and a TermReference on which it will be executed. These actions are stored in a Vector within the Role data structure. It may not be clear at this point, why TermReferences are needed. They are needed because of the existence of term substitutions. The semantics for security protocols makes sure that any variable, which has been set, will be used as a constant from that point on. So taken a definition of role with agent A communicating with agents B and C:

```
role A(B,C)
 var
   nb,nc: nonce;
 def
   read(C, nc);
   read(B, nc);
```

The second read action will only accept a nonce nc, from B, which value is exactly the same as the nonce nc sent by C. This is because the first read action from C causes the variable nc to be substituted by the constant nc.
Terms within the def definition can be arbitrarily complex, consisting of unlimited levels of TermLists and EncTerms. Actions are constructed with BasicTermRefs, retrieved from the symboltable. Since each symbol, representing a variable or constant in the SPDL input file, is linked to exactly one BasicTermRef in the symboltable, whenever a term is set, at any point in the action sequence, it is set throughout the action sequence. Thus using TermReferences facilitates substitutions greatly.
Functions are used in the same manner. Since the value of a function can only be of the basic term type, key, a BasicTermRef can be made for it in the symboltable. One of the limitations of this SPDL implementation is that functions are commutative. Therefore the lookup name in the symboltable can be easily constructed by adding the hashcodes[18] of the function argument names to the hashcode of the function name. It is possible at parse time that one or more argument function value is not known. This can be, because it might depend on a value which needs to be read, or an agent who is a role argument. For that purpose an internal action, fassign (function assignment) has been created. This special action, takes a BasicTermRef, which is linked to by the symboltable, and sets the right function value at runtime. At runtime all arguments will have to be set, either by successive reads or by role argument supplies. If a function with arguments has been declared in a Role's know definition, it is automatically placed as the first sequence of actions of that role. These actions are invisibly executed while the protocol is running. Whenever some Action requires a function value, which has not been explicitly[19] declared in the

[18]A hashcode is an internal representation of an object in the form of an integer. This integer is unique; Objects of the same type, which might have the same value, but which occupy different locations in memory have different hashcodes. Strings with the same value have the same hashcode.

[19]Within the know definition knowledge of complete functions can be expressed by

Role's know definition, this value will first be retrieved by a fassign action.
Roles also keep track of functions in several ways. If a function has been declared locally, the role knows every possible value of that function. This is also the case if there is an implicit declaration of the role's know definition. It is important to keep track of so called complete functions, since if a role conspires with the intruder, every function value possible, will have to be created and transferred to the intruder. If the role knows some function value which has an inverse, it must also keep track of the inverse value. It is very important to keep this value out of the general symboltable. A conspirer can only recognize something which has been encrypted with the inverse of a known function value, but it cannot use that value to encrypt terms. That is the basis of asymmetrical encryption. The implementation of functions is explained in detail in the next subsection.
Runs are constructed from the information gathered by Roles. To create a Run, a Role must be supplied values for its arguments and a run identifier (*runId*). These arguments are gathered from the scenario input file. The *runId* is just a counter, starting with zero and is incremented every time a new Run is spawned. When a Run is created, all the BasicTermRefs in the symboltable of the Role will be deepcloned[20] BasicTermRefs in the symboltable of the Role are cloned, and linked into a new symboltable of the Run. If a BasicTermRef in the symboltable has been set, its target term will also be deepcloned.
Every term in the Role will be renamed to its old name, as it appears in the input SPDL file, concatenated with its *runId*. This renaming makes sure that whenever a trace is read by a human interpreter, differences between terms from the same Role instantiated with the same parameters can still be seen. Runs, like Roles, keep symboltable for the basic terms they use. Because for each new Run, a new symboltable will be created, the Action Vector from the Role cannot be simply copied into the Run. Each action is cloned and the TermReference it contains is mirrored to the new symboltable. Mirroring is done, by comparing the old symboltable to the new and recreating the TermReference in the source Action to reflect the new symboltable.
Functions might need to be deepcloned and mirrored too. Functions which are known completely must only be deepcloned and mirrored if they are declared locally. If they are declared in global or protocol scope, their values can be shared by different Runs and thus won't have to be deepcloned.
Following is a schematic description of the most important methods of these datastructures.

Role			
Name	Return	Parameters	Action
createRun	Run	runId	Creates a Run with runId

definition of that function name without argument (Implicit). Knowledge of one specific function value can be expressed by a definition with arguments.(Explicit)

[20]Deepcloning, contrary to shallow cloning, clones every internal data structure from the target. This insures that clones have no shared objects.

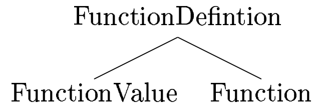
Run			
Name	Return	Parameters	Action
setRun-Parameters	none	Vector params	Sets the ordered Vector of agents as Run parameters.
advanceTo-Read	none	Knowledge K, Set Claims	Executes sends and claim actions until a read action is encountered or all actions have been executed. Updates both Claim and K if needed.
compromise	Set	none	Returns all the knowledge contained in the Run.
assignTerm	none	Term T	Assigns a term to a read action.
nextAction	none	none	Sets the internal action pointer to the next action.
setReady	none	none	Needs to be executed to indicate that the Run has been successfully initiated.

TermReference			
Name	Return	Parameters	Action
getContext	Term	none	Returns a context term which has the same internal structure as the TermReference and holds the right term for every set value of the TermReference or a hole, of the right type, for every unset value of the TermReference.
getTerm	Term	TermReference	Returns the term which has the same internal structure as the TermReference.
clone	TermReference	none	Returns a new TermReference pointing to the same term as the original TermReference.
deepClone	TermReference	none	Returns a new TermReference pointing to a new term, of the same type and name, as the old TermReference points to.
mirror	TermReference	SymbolTable ST	Returns a new TermReference pointing to a term from ST, which has been indexed by the same name as the original term.

Action			
Name	Return	Parameters	Action
getTr	TermReference	none	Returns the TermReference in an action for send and claim actions.
setTr	TermReference	TermReference	Sets the TermReference for read actions.
assign	none	none	Tries to find a value for a function in an internal fassign action.
getFValue	TermReference	none	Returns the value of a function for an internal fassign action.

7.2.4 Functions

The module Functions has the following structure:



Functions are nothing more than a relationship between arguments and a value. In a Function class, these arguments are represented by a set[21] and the function itself as a hashtable. The hashtable links a set to a value, which is a term of the type key. Sets have by definition no order and lend themselves very good to commutative functions, since the order of the arguments is also irrelevant. In order to get a value from a Function, a set of arguments, terms, is supplied to the hashtable. Only if every value of a function needs to be known at runtime, then at some point during initialization a value is created for every possible argument a Role can access. (This is only needed when a compromised agent knows a complete function). If the complete function definition needs doesn't need to be known before runtime, then function values are created on the fly. A Function will, for the same set of arguments, always deliver the same term as value.
When a function declaration is encountered at parse time, the parser has no knowledge about possible values of its arguments. It only knows the definition. for each function declaration a FunctionDefinition object is first created. When a function value is needed, either within the know definition or within a def definition, a FunctionValue object is created, which supplied TermReferences to its arguments. This is needed because the value of an argument might not be known at parse time[22]. FunctionValue objects are then used to create fassign actions. When these actions are executed, all TermReferences in set of function arguments, have been set (or an error will occur) and a function value can be created or retrieved.
Following is a schematic description of the most important methods of these datastructures.

Function			
Name	Return	Parameters	Action
getTerm	Term	Set Params,Int type	Returns a term as value for those the set of parameters. If two sets of parameters are different, then two different terms will be returned by getTerm.
mirror	Function	SymbolTable ST	Returns a new function, which has copied all the values from the Symboltable.

[21]Instead of the Java class java.util.Set java.util.Hashset is used, because it offers more functionality.

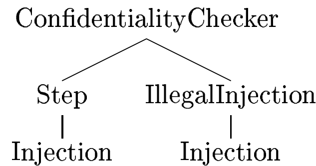
[22]Remember that some TermReferences will not be set until initialization (arguments for Roles), or even execution (TermReferences set by read actions)

FunctionDefinition			
Name	Return	Parameters	Action
fillFunction	none	Set instances	Fills the function by pairing parameters with values. The parameters are generated by combining terms found in instances according to the functionDefinition's type.
extract-FunctionValues	Set	none	Returns the set of terms constituting the range of the function.
clone	Function-Definition	none	Returns a shallow clone.
deepClone	Function-Definition	none	Returns a deep clone, in which for every function value of the original a new value is created with the same type.

FunctionValue			
Name	Return	Parameters	Action
addParam	none	TermReference Tr	Adds TR to the list of parameters.
getValue	Term	none	Returns a term which is the value of this FunctionValue Function for its parameters.
mirror	FunctionValue	SymbolTable Functions, SymbolTable Terms	Returns a new functionValue object which has been mirrored using the symboltables.

7.2.5 ConfidentialityChecker

The ConfidentialityChecker consists of four parts. Three of them are used to support the algorithm in the main part.



These three parts are:

- Injection
  An Injection is a datastructure that contains enough information to be used as a semantic injection. An Injection keeps administration of one *runId* and a message. If this Injection would be performed, the term which is stored in the message will be injected to a Run, identified by the *runId*.

- IllegalInjections
  IllegalInjections is a datastructure, which is used to store those Injections, which may not be used at a certain state. For each Run which has illegal Injections, it keeps a list which can contain multiple Injections.

- Step
  A step is a collection of Injections. These Injections will be performed when the step is executed. There can only be one Injection to any Run in one Step. A Step can also contain a collection of illegal Injections. These Injections may not be performed in the next state. When a step is executed, all Runs which have Injections in that step will be updated. Executing a Step can have a side effect on the IllegalInjections datastructure. Illegal Injections inside a Step, for Runs which will not be receiving an Injection from this step, will be added to the IllegalInjections datastructure of the next state.

- IllegalInjections
  The datastructure IllegalInjections is used to keep track of those Injections which may not be executed in some state. There is a difference between the datastructure IllegalInjections and the illegal Injections contained in a step. A step can be seen as a recipe for a state, it indicates what will change in a state when the step is executed. IllegalInjections is part of the state. When a step is executed its illegal Injections will be added to the states IllegalInjection datastructure. It may contain illegal Injections as a result of previous states.

The ConfidentialityChecker algorithm is implemented by the main Module, ConfCheck. This recursive algorithm accepts a model and gives a verdict on whether or not the secrecy security goal has been met. If it does find a security breach it provides a trace. The algorithm can be run with several options, which can be specified by to the controller by executing it with special flags.

Following is a schematic description of the most important methods of these datastructures.

Injection			
Name	Return	Parameters	Action
getMessage	Term	none	Returns a term which is to be injected.
getRid	Integer	none	Returns the runId of the run, which is the target of this Injection.

Step			
Name	Return	Parameters	Action
addInjection	none	Injection i	Adds Injection i to the set of Injections.
setIllegal	none	Set I	Sets the set of Injections which are illegal.
remove-Injection	none	Injection i	Removes an illegal Injection from the set of illegal Injections.
getInjection	Injection	Integer runId	Returns the Injection for a Run identified by runId.
getInjections	Set	none	Returns the set of Injections.
getIllegal	Set	none	Returns the set of illegal Injections.

IllegalInjections			
Name	Return	Parameters	Action
addInjection	none	Injection i	Adds i to the set of illegal Injections.
clone	IllegalInjections	none	Returns a new Illegal-Injections datastructure containing copies of the original Injections.
isIllegal-Injection	boolean	Injection i	Returns true if i belongs to the set of illegal Injections.
isIllegal-Injection	boolean	Term t, Integer runId	Returns true if injecting *t* to a Run identified by runId is an illegal Injection.
removeIllegal	none	Injection i	Removes i from the set of illegal Injections.

7.2.6 Controller (Ccheck)

The controller is the command line interface used by the end user. It uses user input for initialization and starts the model-checking process. This name of the controller is Ccheck. Check is executed by using the Java virtual machine. Assuming the class-path has been set to include all modules the syntax is

```
\$ java [-dcsft] Check spdlfile scenariofile
```

Where [-dcsft] constitutes one of the following flags:

- `-d`

  The debug flag, will cause the algorithm to run in debug mode. Before each state, the contents of the intruder knowledge, the illegal Injections and all possible steps for that state are printed to the standard output stream.

- `-c`

  The continue flag forces the algorithm to continue checking after it finds a hole. This is used in order to find all the holes in a model.

- `-s`

  The sign of life flag will cause a dot character to be printed to the standard error stream, every time a new state is reached. Model-checking can take a while and without some sign of life, it might seem the program has crashed.

- `-f`

  The flash flag is used as a special debug tool. It flashes by printing an asterisk to the standard error stream, whenever a claim has been verified. It was found useful in constructing SPDL-scripts in order to make sure, at least one trace reaches some point.

- `-t`

  The trace flag is used to print a trace when ever a state, which has no next step, is reached. If used in conjuncture with the continue flag, it will print all possible traces.

- `-r`

  The run flag will cause the controller to print out all initialized runs.

7.3 The initialization process

Initialization and execution is done by a controller program called Ccheck. Apart from initialization of the appropriate datastructures Ccheck, also handles and sets flags received from the command line.
To the user of the SPDL model-checker, a model consists of a SPDL input file and a scenario file. The model, the confidentiality checking algorithm accepts, consists of a set of uncompromised Runs, a set of secrecy claims, the intruder knowledge and the set of illegal Injections. The initial set of uncompromised Runs consists of the Roles instantiated in the scenario input file, which do not have an agent controlling the Run which has been defined as untrusted. Although Runs will change while the algorithm is executed, the composition of this set will always remain the same, throughout the execution of the algorithm.
The initial set of secrecy claims consists of all claims from all Runs which have been defined before the definition of the respective Run's first read operation. For each term claimed secret, administration is kept, in the form of a *runId*, about the origin of the claim.
The initial intruder knowledge consists of one unique intruder term, for each of the existing basic terms types and all knowledge of all the compromised agents. This knowledge is kept in a Knowledge object. The initial set of illegal Injections is empty. Illegal Injections are kept in an IllegalInjections object. The process of creating and filling the appropriate datastructures from the two input files takes three stages.

Stage one parses the SPDL input file
The main purpose of this process is to create Roles. In order to do this, function and variable/constant symboltables are kept for each scope. (Local symboltables

are stored within a Role) Roles are furthermore organized by the protocol in which they are defined.

Stage two parses scenario input file
While parsing the scenario file, the application gathers all the information it needs in order to assemble the datastructures which will be used as the input model for the confidentiality-checking algorithm.
Runs are created from Roles, following the expressions in the scenario file. Special administration separates compromised Roles from uncompromised ones. This fills the following structures

Stage three creates initial intruder knowledge
The basic knowledge the intruder starts out with, consists of one unique term, from each of the defined basic term types. These terms are not known to any of the defined Runs. Each agent, trusted and untrusted, specified in the scenario file will also always be included in the intruder knowledge.
Next every Run will be inspected for Functions that need to be filled out completely. Each function is checked for its scope and parameter composition. Depending on its scope a set of terms for each function parameter is constructed. This set is increased by the elements in the intruder knowledge. Each function parameter might have several options. By combining these parameter sets, every possible function parameter is generated. Subsequently for each parameter, a value is stored in the function. Combining parameters in order to span entire function domains is costly. Users should therefore keep full function knowledge of agents to a minimum. On the other hand, since this is done before the actual model-checking, the problem is of limited concern.
If a Function has been generated and it has an inverse, then its inverse must also be generated and the values of those functions, which are by definition keys, will be linked.
After function generation, every Run has all the knowledge it needs to perform its Role. Now Runs, controlled by conspirating agents can be compromised. Compromising a Run will result in all its knowledge to be transferred to the intruder.

At this point the model defined by the two input files, has been translated into a set of datastructures, which can be used by the model-checking algorithm.

8 Validation

The SPDL secrecy model-checker is able to check real life security protocols for holes. In order to demonstrate and evaluate this ability, all protocols in SPORE, which can be expressed in SPDL and insure secrecy were used as a test collection.

Protocol	Attacks	Found	Time (ms)	States/Mini states
Andrew Secure RPC	compromised key	no	2724	10522/20036
BAN modified Andrew Secure RPC	compromised key	no	2824	10594/20110
BAN concrete Anderw Secure RPC	none	no	2053	11901/28985
Lowe modified BAN concrete Andrew Secure RPC	none	no	2053	11890/28961
Kao Chow Authentication v.1	compromised key	no	11357	18696/41914
Kao Chow Authentication v.2	none	no	55470	18696/41914
Needham-Schroeder Public Key	parallel session	yes	80	58/177
Lowe's fixed version of Needham-Schroeder Public Key	none	no	5508	14312/38008
Otway Rees	typeflaw	no	6389	3876/11016
TMN	Authentication failure	yes	50	11/30
Woo and Lam Mutual Authentication	parallel session			
Yaholom	none	no	3405	715/2013
BAN simplified version of Yaholom	replay + typeflaw	no	12868	2495/7593
Lowe's modified version of Yaholom	none	no	2103	1360/3888
Paulson's strengthened version of Yaholom	none	no	12789	2495/7593

9 Evaluation

This section deals with the evaluation of choices made to realize the model-checker. The first choice made was the approach to build the checker. The bottom-up way of creating this model-checker proved to work quite well. Since a clear and formal semantic was available, the major parts needed for the actual model-checking were known early in the project. After enough objects had been built to support the semantics for security protocols and the `checkSecrecy` algorithm had been designed, building was a straightforward job. There where, however, some hurdles encountered, which might not have existed if another building approach was taken. For example, symmetrical, encryption was taken as a starting point. When later in the development asymmetrical encryption had to be supported, a lot of parts in the existing program had to be patched.
The entire model-checker was written in Java. The parsers had been generated using JavaCC. The language Java made it possible to create this model-checker in such a short time. Java provides numerous datastructures which can be used for all kinds of administration. The Java datastructures which proved to be most useful where java.util.HashSet, java.util.Hashtable and java.util.Vector, which respectively are

used for unordered storage of datastructures, the creation of lookup tables and the ordered storage of datastructures. These datastructures provide numerous methods which made building in Java very efficient. Drawbacks of Java come in its performance. Because of the general nature of the used Java objects they might be inefficient. JavaCC provided a very good interface to create the parsers. There were no noticeable drawbacks to using JavaCC.
Most of the implemented algorithms are very compact and straightforward. Every module built was tested thoroughly before it was integrated in the system. Some bugs, however, did manage to find there way into the system. There is a major memory leak in the `checkSecrecy` algorithm. This is caused by the strategy chosen to find transitions in a state. In every state all possible transitions are calculated. If there are enough runs instantiated, which yield enough options, the composition of these options can totally fill memory. The effect is that this routine consumes so much memory, that the operating system starts using virtual memory[23] which causes the program to become very slow. After a while, Java will detect the abnormal memory consumption and will generate an OutOfMemmory Exception and the whole system crashes. Because of this leak, this version of the SPDL secrecy Model-checker cannot be used to check very big scenarios. A possible solution of this problem is presented in the next section.
Another algorithm that might still contain bugs is the FillContext algorithm in the Knowledge module. FillContext uses an implementation of the unify function as defined in the semantics for security protocols. Although the effects of this function has been described in detail in the current semantics for security protocols and there exists a formal method for unify on untyped terms, there still is no formal method known which exhibits the behavior for typed terms. FillContext was very hard to create and might still contain bugs, in the form of missed traces. FillContext is the bottleneck of the model-checker, where support for attacks must be created. It was created using a spiral prototyping engineering method. This means that a prototype is built, tested against real life protocols which contain known attacks and altered if it missed an attack on the protocol. All implementable protocols of SPORE were used and many spirals were made. However, this does not ensure that all attacks can be found with the latest version of FillContext. The only way for a correct implementation of FillContext to be realized is a complete re-write based on explicit formal semantics for security protocols.
Performance wise, the program runs protocols with limited scenarios fast. As the scenarios tend to get bigger, the program gets exponentially slower. This too, is the effect of the bug in FillContext, which will reserve ever more memory to store substates.
Although the program has been written in Java and does work on any O.S platform which supports the Java virtual machine there are platform inconsistencies. These inconsistencies occur in the order of the traces presented. This is caused by the usage of HashSets. These sets do not guarantee ordering. This has, however, no effect on the actual verdict of the program.

[23]Virtual memory is created by using part of the harddisk as memory.

10 Future work

As explained in Section 4.3 some limitations have been formulated in order to make this project feasible within the given time constraints. Future editions of the model-checker can incorporate other security goals. As for the secrecy goal, it must be possible to design the use of compromised keys by the intruder. This model-checker only supports typed communication. If Roles can accept any message instead of some typed message, type-flaw attacks can be expressed. Type-flaw attacks work by sending a term to an agent, which is not the type it expects. Type-flaws support should be implemented with various gradations, because it will exponentially increase the state space. Type-flaws will have to be implemented by augmenting the input language and the FillContext function.
The two known problems of the current version lie in the OutOfMemmory Exception and the FillContext function. The OutOfMemmory Exception can only be solved if a completely other approach is taken in the `checkSecrecy` algorithm. In the current approach the entire supertransition space of a state is computed when a state is reached. Each of these supertransitions is then checked one by one. The OutOfMemmory Exception occurs when the transition space becomes too big to contain in main memory. Instead of this approach, some kind of pipelined approach can be created, in which while transitions are computed subtransitions are also checked. In this way, when transition $t_1 \rightarrow run_1, ..., t_n \rightarrow run_n$ is performed every sub transition has already been performed and removed from memory. This will require a major rewrite of the `checkSecrecy` algorithm.
FillContext also needs to be redesigned and rewritten in order to implement a formal specification of the unify action.
Platform inconsistencies, with regard to trace ordering, can be avoided by using ordered sets, like Vectors.

11 Learning points

Building the secrecy model-checker has made me dig deeper into various computer science subjects. Computer security has always been a hobby of me, but I never experimented with it on a formal level. Figuring out how the semantics for security protocols work was the first learning point in this project for me. It took various hand and paper examples to get some feeling of its usage.
Before this project I had very limited experience with model-checking. This project forced me to find out more about what it is about and why there is so much research done in this field.
On the practical side, I've been programming for years, but never written an interpreter. Figuring out JavaCC and building the parsers with a foundation of Computer Science will be a skill, I think I'll be using in the future.

12 Conclusion

The semantics for security protocols which is currently in research by the Security Group of the Eindhoven University of Technology can be used to create a model-checker for the secrecy security goal. The SPDL secrecy model-checker supports parallel session attacks and replay attacks and might be modified to support com-

promised key attacks. Type-flaw attacks, however, will prove to be more of a hassle to support. Choosing a scenario is crucial in finding possible holes in a protocol. Overspecifying will create enormous state spaces and the application will run out of memory. However, even with this existing bug, it was possible to check a great deal of the security protocols available in SPORE. If a good choice has been made for a scenario, the application performs quite reasonably. Bad choices are severely punished.
The main drawback of this implementation lies in its performance. The application could work much faster, if it was written in a more low level language. Memory consumption, which is the major problem of this application, can be fixed by adapting a part of the main algorithm.
When there is formal description in the semantics for security protocols on the realization of unify available, this implementation can be easily modified to support this description. This will make it far more stable.

13 Appendix A

This appendix contains the scenario-files which were used in the evaluation of the model-checker. Only those files that lead to a security bug in a protocol are displayed here. (All files are included in the distribution of the model-checker.)

13.1 Needham-Schröder public key protocol

```
scenario

agent
trusted: A,B,S;
untrusted: M;

run
NSPK.A:A(B,S);
NSPK.B:B(A,S);
NSPK.S:S(A,M);
NSPK.S:S(A,B);
NSPK.A:M(B,S);
NSPK.A:A(M,S);
```

Trace:

```
Security hole found


******* Failed terms *******
TL(nb#1 na#5) claimed secret by NSPK.B:B(A,S)@1


******* Action trace which breaks the protocol: *******
0 SEND   [NSPK.A:A(M,S)@5] A -> S: TL(A M)
1 SEND   [NSPK.A:A(B,S)@0] A -> S: TL(A B)
```

```
2 INJECT [NSPK.S:S(A,M)@2] A -> S: TL(A M)
3 INJECT [NSPK.S:S(A,B)@3] A -> S: TL(A B)
4 SEND   [NSPK.S:S(A,B)@3] S -> A: ENC(TL(KP(B) B),KS(S))
5 SEND   [NSPK.S:S(A,M)@2] S -> A: ENC(TL(KP(M) M),KS(S))
6 INJECT [NSPK.A:A(M,S)@5] S -> A: ENC(TL(KP(M) M),KS(S))
7 INJECT [NSPK.S:S(A,M)@2] M -> S: TL(M A)
8 SEND   [NSPK.A:A(M,S)@5] A -> M: ENC(TL(na#5 A),KP(M))
9 SEND   [NSPK.S:S(A,M)@2] S -> M: ENC(TL(KP(A) A),KS(S))
10 INJECT [NSPK.B:B(A,S)@1] A -> B: ENC(TL(na#5 A),KP(B))
11 SEND   [NSPK.B:B(A,S)@1] B -> S: TL(B A)
12 INJECT [NSPK.B:B(A,S)@1] S -> B: ENC(TL(KP(A) A),KS(S))
13 SEND   [NSPK.B:B(A,S)@1] B -> A: ENC(TL(na#5 nb#1),KP(A))
14 CLAIM  [NSPK.B:B(A,S)@1] B: SECRET TL(nb#1 na#5)
15 INJECT [NSPK.A:A(M,S)@5] M -> A: ENC(TL(na#5 nb#1),KP(A))
16 SEND   [NSPK.A:A(M,S)@5] A -> M: ENC(nb#1,KP(M))

*******Statistics*******
Visited: 17 detected miniStates: 95
Checking security took 40 miliseconds
```

13.2 TMN protocol

```
scenario

agent
trusted: A,B,S;
untrusted: M;

run
//Minimal input to reconstruct attack 1 of spore:
tmn.A:M(B,S);
tmn.B:B(A,S);
tmn.S:S(A,B);
```

Trace:

```
Security hole found


******* Failed terms *******
TL(kb#1) claimed secret by tmn.B:B(A,S)@1


******* Action trace which breaks the protocol: *******
0 INJECT [tmn.S:S(A,B)@2] A -> S: TL(B ENC(intruderKey,PK(S)))
1 INJECT [tmn.B:B(A,S)@1] S -> B: A
2 SEND   [tmn.B:B(A,S)@1] B -> S: TL(A ENC(kb#1,PK(S)))
3 CLAIM  [tmn.B:B(A,S)@1] B: SECRET TL(kb#1)
4 SEND   [tmn.S:S(A,B)@2] S -> B: A
5 INJECT [tmn.S:S(A,B)@2] B -> S: TL(A ENC(kb#1,PK(S)))
```

```
6 SEND   [tmn.S:S(A,B)@2] S -> A: TL(B ENC(kb#1,intruderKey))


*******Statistics*******
Visited: 11 detected miniStates: 30
Checking security took 20 miliseconds
```

References

[1] E.P. de Vink C.J.F. Cremers, S. Mauw. Operational semantics of security protocols, in preparation.

[2] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, March 1983.

[3] Dieter Gollmann. *Computer Security.* John Wiley & Sons ltd, England, 1999.

[4] I. Kao and R. Chow. An efficient and secure authentication protocol using uncertified keys. *Operating Systems Review*, 29:14–21, July 1995.

[5] Lamport, Shostak, and Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press.* 1995.

[6] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

[7] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.