

Efficient Checking for Parallel Attacks in Casper

Tilo Buschmann
Keble College, University of Oxford



Handed in: 01. September 2006
Supervisor: Prof. Bill Roscoe

Efficient Checking for Parallel Attacks in Casper

Dissertation

at the
University of Oxford

Tilo Buschmann, Keble College

Abstract

Security protocols are sets of rules to establish security properties between interacting entities, including secrecy, authentication, anonymity and non-repudiation. They are widely used for securing communication and therefore need to undergo rigorous testing and verification.

The basis for the verification of security protocols in this dissertation is the process algebra CSP, which can be used to model security protocols as networks of trustworthy agents and the intruder. The CSP model checker FDR is used to test specified assertions of these protocols.

The original CSP models of security protocols is not complete, i.e. it can find attacks upon protocols, but is not able to prove the absence of flaws under the Dolev-Yao assumption. The Data Independence model of [6] provides some completeness but tends to take very long, for some protocols hours or days.

Kleiner and Roscoe developed an unbounded model in [4], that uses internalised agents and a collapsing function to provide a finite and complete model to check security protocols. In other words, it proves the correctness of a protocol if it does not find an attack. At the same time, it promises to reduce the number of checked states considerably, therefore taking less time.

Casper is a piece of software that compiles descriptions of security protocols into a CSP model. This dissertation provides an implementation of the new unbounded model as an extension to Casper. Furthermore, the implementation is tested with well known protocols.

A drawback of the model is the increase of necessary calculations, which was overcome using several optimisation techniques.

The benchmark results of the final implementation show that the new implementation conveys parallel attacks upon security protocols reliably and brings also a considerable decrease in checked states and compilation calculations, not only in comparison to the model of [6] but also in comparison to the original, non-proving model.

Acknowledgements

I thank my supervisor Bill Roscoe for introducing me to this wonderful subject and his guidance throughout the dissertation. A special thank goes to Eldar Kleiner, who provided much insight and helped me with many fruitful discussions. This dissertation would not have been possible without him. I also thank David Walker for his exceptional analytical abilities and his support during the harder times of my MSc.

Furthermore, I thank Sara Adams for proof reading my dissertation and giving me insight into higher mathematical and typographical standards.

Finally I thank my family as well as Benjamin Damet, Angelina Davydova, and my circle of friends in Oxford for supporting and encouraging me during my MSc.

Contents

1	Introduction	1
2	Background	3
2.1	Security Protocols	3
2.2	CSP	5
2.2.1	Basic Notation	5
2.2.2	Parallel Operators	6
2.2.3	Hiding and Renaming	7
2.2.4	Traces	8
2.2.5	FDR as a Specification Checking Tool.	9
2.3	The Bounded CSP Model for Checking Security Protocols	10
2.3.1	Data Types	10
2.3.2	Trustworthy Agents	11
2.3.3	The Intruder	12
2.3.4	Connecting the Components	14
2.3.5	Specifying Protocol Goals	15
2.4	The Unbounded Parallel model	17
2.4.1	Data Independence	18
2.4.2	The Collapsing Function	19
2.4.3	Internalising Agents	20
2.4.4	Secrecy Analysis	21
2.4.5	Authentication Analysis	22
3	Casper implementation of the Unbound Parallel model	23
3.1	Changes to the Casper Scripts	23
3.1.1	The #Processes Section	23
3.1.2	The #Specification Section	24
3.1.3	The #Actual variables Section	24
3.1.4	The #System Section	25
3.1.5	The #Intruder Information section	26
3.1.6	Correct Configuration for Different Checks	26
3.2	Construction of the Internal Agents	27
3.2.1	Implementing the Collapsing Function	27
3.2.2	Building the Deductions	29
3.3	Secrecy Decision	34
3.4	Type Checks and Consistency	36
3.4.1	Additional Checks for the Original Model	36
3.4.2	New Checks for the Unbounded Parallel Model	36
3.4.3	Unimplemented Checks	37
3.4.4	Unimplemented Warnings	37

4	Optimisations	39
4.1	Decreasing the Number of Learnable Facts	39
4.2	Decreasing the Size of the External Agents	42
4.3	Excluding the Identity of the Intruder from External Protocol Runs	44
4.4	Decreasing the Number of Renamings	45
4.5	Minimising Fact ₁	46
5	Benchmark results	48
5.1	Legend	48
5.2	Results	49
5.2.1	Yahalom Protocol	49
5.2.2	Needham/Schroeder Public Key Protocol	50
5.2.3	Needham/Schroeder/Lowe Public Key Protocol	50
5.2.4	BAN simplified version of the Yahalom protocol	51
5.2.5	Otway Rees Protocol	52
5.2.6	Neumann Stubblebine	52
5.2.7	TMN Protocol	53
5.3	Discussion	54
6	Discussion	56
6.1	Future Work	56
6.2	Conclusion	58
A	Casper Input rcripts	59
A.1	Bounded Model Yahalom Protocol Specification	59
A.2	Extended Yahalom Protocol	60
B	Yahalom CSP Script	61
C	Excerpt from the Casper Extension	76
	References	78

1 Introduction

The ubiquitous use of computers in daily life increases the risk to sensitive information and the personal belongings of people and organisations who use them. Governmental agencies, financial institutes, the military, universities, and ordinary people rely on the untampered use of modern technologies.

For example, people get convenient access to the functions of their bank accounts through the Internet. All transactions between the user and the bank, as well as the transactions between banks (e.g. transfers by wire) are secured using cryptographic methods. The large amounts of money that are involved obviously need to be protected.

Therefore, widespread efforts are made to guarantee the reliability and security of these systems. Universities and industry develop and research better technologies, which can be seen by the number of publications and the worth of the market in this field.

Unfortunately, security holes and flaws in security technologies are found on a regular basis, which leads to an increased number of hackers and scammers, who exploit them. Consequently, it is desired to check and verify the security measures.

The topic of this dissertation is the examination of security protocols, as means of achieving secrecy, authentication, anonymity and non-repudiation. The question of the reliability of cryptography itself is not touched, the correctness will even be defined as unbreakable throughout this thesis.

The basis of this thesis is the process algebra CSP, which can be used to model concurrent systems and in this case security protocols. It offers the means to specify the exact behaviour of agents within a network, the intruder, that tries to break the assertions of the security protocol as well as the assertions themselves. The software FDR can be used to verify these assertions.

As the feature set, checks and general requirements of security protocols are very diverse, it is quite uncomfortable for the researcher to write a new CSP description for every security protocol. Therefore, Casper was written, a compiler that takes a description of a security protocol and generates the correct CSP script to be tested with FDR.

Two main obstacles had to be surmounted: FDR can only check finite models, i.e. it is not possible to work with an unlimited amount of values, that are often required by security protocols (e.g. nonces or session keys). Furthermore, the original CSP model is bounded, i.e. it works with a restricted number of agents within the simulated network and might find attacks. But the model is not able to prove the absence of security attacks under the Dolev-Yao assumption.

For this reason, unbounded models were developed. One of them, the Data Independence model developed in [6], provides a more complete analysis of security protocols. Unfortunately, the model has been shown to be quite slow, taking several hours for some protocols, or even days.

Kleiner and Roscoe proposed a new model in [4] that promised to be both complete and efficient. It was extended in [3] to include further optimisations.

Yet, Casper does not produce CSP scripts using this new method and the general usability and efficiency of the model is not known. The goal of this dissertation is the implementation of the Kleiner model in Casper, to optimise the model and to test the efficiency and ability to find flaws in protocols.

The object of this project was to automate this new model within Casper. In this thesis we report on how this was accomplished and on the results obtained.

2 Background

2.1 Security Protocols

Definition

Protocols are sets of rules for the interaction of different entities that try to achieve certain objectives. In the case of *security protocols*, these objectives can be the secret transport of an encryption key, the authentication of one agent to another, non-repudiation, anonymity, integrity, or other goals.

Security protocols can be described by the sequence of the messages that are exchanged between the involved entities and by the structure of individual messages. A security protocol will usually use cryptography for reaching its goals, this includes the usage of symmetric and asymmetric encryption, hash functions, and digital signatures.

The Yahalom Protocol

The following example defines a variant of the Yahalom protocol in the notation that is commonly used in literature:

Message 1. $a \rightarrow b : na$
 Message 2. $b \rightarrow s : \{a, na, nb\}_{ServerKey(b)}$
 Message 3a. $s \rightarrow a : \{b, kab, na, nb\}_{ServerKey(a)}$
 Message 3b. $s \rightarrow b : \{a, kab\}_{ServerKey(b)}$
 Message 4. $a \rightarrow b : \{nb\}_{kab}$

Every line denotes one message of the protocol. The left side of the colon defines the sender and receiver of the message. Message 1 therefore is sent from agent a to agent b . The right side of the colon defines the content of the message. In message 4, the nonce nb is sent, encrypted with the session key kab , that was newly generated in this protocol run.

The content of messages consist of atomic terms and recursively built structures.

The following atomic terms can appear, for example:

- a, b, s - the identity of entities involved in the protocol run
- na, nb - nonces, unique and unpredictable numbers that were freshly generated for the protocol run
- $kab, ServerKey(a)$ - symmetric keys, used to encrypt messages in the Yahalom protocol

Other atomic items might be used, but the ones above are the most common and will be referred to in this dissertation frequently.

Messages are constructed in the following way:

- Atomic terms are messages

- Concatenation of messages are messages, written as X,Y for a concatenation of X and Y
- Encrypted messages are messages, written as $\{X\}_K$, where X is encrypted with K

The list may be expanded, if further cryptographic and hashing methods are used. For the purpose of this dissertation, the above limited definition is sufficient. More details may be found in [7].

In the above protocol, message 1 contains a single nonce, that was newly generated by agent a . Agent b then sends the identity of a together with the previously received nonce and a newly generated nonce nb encrypted with $ServerKey(b)$ to the server s in message 2. $ServerKey(b)$ is defined as a symmetric key, that is possessed by both the server s and b , i.e. s and b can encrypt and decrypt messages with it. The agent a has a similar key, that he shares with s . The server s then sends messages 3a and 3b with a newly generated session key kab and a sends the last message 4 to be with the nonce nb encrypted with kab .

Flaws in Security Protocols

Flaws in security protocols are found even years after they have been introduced and sometimes checked or proven. One example is the Needham-Schroeder Public Key protocol, that was found to be flawed by Lowe in [5], using similar methods to the one described in this chapter.

The following example is a variant of the Needham-Schroeder Public Key protocol:

Message 1. $a \rightarrow b : \{na\}_{PK(b)}$
 Message 2. $b \rightarrow a : \{na, nb\}_{PK(a)}$
 Message 3. $a \rightarrow b : \{nb\}_{PK(b)}$

The protocol uses asymmetric encryption, $PK(a)$ is the public key of a , anyone can use this key to encrypt messages for a , which only a can decrypt.

The following is a successful attack in a similar notation to the protocol description used above. For the notation of an actual run of the protocol, the variables are substituted with real values, which will be started with a capital letter throughout this dissertation:

$\alpha 1.$ Alice \rightarrow Ivo $: \{Na\}_{PK(Ivo)}$
 $\beta 1.$ Ivo_{Alice} \rightarrow Bob $: \{Na\}_{PK(Bob)}$
 $\beta 2.$ Bob \rightarrow Ivo_{Alice} $: \{Na, Nb\}_{PK(Alice)}$
 $\alpha 2.$ Ivo \rightarrow Alice $: \{Na, Nb\}_{PK(Alice)}$
 $\alpha 3.$ Alice \rightarrow Ivo $: \{Nb\}_{PK(Bob)}$
 $\beta 3.$ Ivo_{Alice} \rightarrow Bob $: \{Nb\}_{PK(Bob)}$

The identity Ivo_{Alice} means, that an intruder with the identity Ivo pretends to be $Alice$. The above description is a protocol run. $\alpha 1$ is message 1 in the first session, while $\beta 1$ is message 1 of a second session.

In the above attack, agent *Alice* starts a session with the intruder *Ivo*. The intruder thereupon starts a new session with *Bob*, in which the intruder pretends be *Alice*. The intruder forwards the nonce from message 1 from the first session to *Bob* in the second session. He also forwards the response (message 2) that he gets in the second session to *Alice* in the first session and then receives the nonce *Nb* encrypted with the public key of the intruder (message 3).

This is a serious security flaw in the protocol, because the intruder gets access to the nonce *Nb* of the second session and is authenticated as *Alice* to *Bob*.

2.2 CSP

CSP (*Communicating Sequential Processes*) is a process algebra that is used to describe processes which interact both with the environment and with possibly other parallel processes.

CSP is very well suited for specifying, designing as well as, with the help of FDR, checking concurrent systems. There is a well established mathematical theory behind model proving with FDR and CSP, which makes FDR a very usable tool to check security protocols.

2.2.1 Basic Notation

CSP processes interact with the environment or other processes through events. The set of all events, that is used in the communication is the alphabet, which is called Σ . A process is defined by the way it accepts or does not accept events.

The two simplest processes are STOP and SKIP. Both represent processes that are terminated and do not allow any further events. In particular, STOP represents a deadlock, while SKIP represents the successful termination.

These processes are very useful for the modelling of more complex processes and specification.

The following is a very basic process in *prefix*-notation:

$$\text{COFFEAUTO1} \hat{=} \text{coin} \rightarrow \text{coffee} \rightarrow \text{STOP}$$

The process definition can be read in the following way: The process accepts the event *coin*, then the event *coffee*, and then the process behaves like STOP. That example gives a very simple coffee automaton, which stops after first serving. Processes can also be defined recursively, which gives a more usable coffee automaton:

$$\text{COFFEAUTO2} \hat{=} \text{coin} \rightarrow \text{coffee} \rightarrow \text{COFFEAUTO2}$$

The coffee automata accepts a coin, gives a coffee and then behaves like itself again, therefore being able to serve again.

An event can also be defined to contain data. For this, in [7], an event is described as starting with a channel name, that can be followed by data components. A data component is concatenated to the channel with a period, whereas a question mark indicates an input and an exclamation mark indicates an output.

The following is an example of a copy process:

$$\text{COPY} \hat{=} \text{left}?x : T \rightarrow \text{right}!x \rightarrow \text{COPY}$$

In the above example, the environment can decide to communicate an event $\text{left}.x$, with $x \in T$, followed by the event $\text{right}.x$, where the x is bound to the value of the previous left-event.

This channel concept will later be used to have common send and receive events for the network model, that is used in the security check, i.e. it allows to have a send event in the form $\text{send}.Alice.Bob.Na$, where *Alice* sends the message *Na* to *Bob*.

It is possible to give the environment a choice of events to perform next. The external choice operator \square is used as:

$$P1 \hat{=} Q \square R$$

The environment has the choice between the initial events of the process Q and R . An example is the following automaton, that can serve coffee or tea:

$$\text{AUTOMATON1} \hat{=} \text{coin} \rightarrow (\text{coffee} \rightarrow \text{AUTOMATON1} \square \text{tea} \rightarrow \text{AUTOMATON1})$$

Another choice operator lets the environment choose any event x from a subset X of Σ . The following example process accepts such an initial choice and then behaves like $P3(x)$.

$$P2 \hat{=} ?x : X \rightarrow P3(x)$$

The internal choice operator \sqcap allows the nondeterministic choice between processes. This means, that the process chooses internally, which of the two decisions is to be made and the environment gets only this choice presented, thus having no means of influencing which event it can communicate.

For example, suppose the process is defined as:

$$(a \rightarrow \text{STOP}) \sqcap (b \rightarrow \text{STOP})$$

After choosing the left-hand choice, the environment can only communicate the event a and the event b is refused to the environment.

2.2.2 Parallel Operators

The operators described until now are sufficient to build simple sequential processes. But one of the strengths of CSP is the ability to put several processes into parallel and let them communicate with each other.

The strictest parallel operator is $X \parallel Y$, which is used to synchronise two processes on every event, $P \parallel Q$ can perform an event $a \in \Sigma$ only when they are offered by P as well as by Q. The following process is an example:

$$P4 \hat{=} (?x : X \rightarrow P(x)) \parallel (?y : Y \rightarrow Q(y))$$

The process will allow an initial event from the $X \cap Y$ and then behaves like $P(x) \parallel Q(y)$ (with $x = y$). If $X \cap Y = \emptyset$, then P4 will behave like STOP.

The process can also be unwound:

$$P4 \hat{=} ?x : X \cap Y \rightarrow (P(x) \parallel Q(x))$$

A more flexible parallel operator is the *interfaced* parallel operator:

$$P \parallel_X Q$$

This process synchronises only on the events $X \subset \Sigma$, while all other events may be communicated freely.

The following example is taken from [8]. Let $P = ?x : A \rightarrow P'(x)$ and $Q = ?x : B \rightarrow Q'(x)$, so that:

$$\begin{aligned} P \parallel_X Q &\hat{=} ?x \in X \cap A \cap B \rightarrow (P'(x) \parallel_X Q'(x)) \\ &\square ?x \in A \setminus X \rightarrow (P'(x) \parallel_X Q) \\ &\square ?x \in B \setminus X \rightarrow (P \parallel_X Q'(x)) \end{aligned}$$

The process $P \parallel\parallel Q$ interleaves P and Q, they are not synchronised on any events. On the other hand, the process

$$P \parallel_{A \cap B} Q$$

synchronises on any events, that are in $A \cap B$ and does not synchronise on any other events.

It is apparent, that these parallel operators give the means to model agents of a network and their communication. This will be expanded in detail in Section 2.3.

2.2.3 Hiding and Renaming

In [1] Broadfoot outlines, how it is desirable to change the appearance, i.e. the visibility of events of a process P to outside. In this sense, the events communicated by P are divided into events that can be seen from the outside and those events, that are internal to the process P.

There are two important operators in CSP, that can be used to influence the visibility of events to the environment. These can be used to build sub-components, that run in parallel and who communicate with other components only over certain designated events, while the internal behaviour is hidden.

The hiding of events in CSP is done using the hiding operator. The notation $P \setminus E$ hides any events $e \in E$ in the process P.

Another useful operator is the renaming operator. The process $P [[R]]$ behaves like P, but renames all events of P according to the renaming relation R.

The coffee and tea automata could for example be renamed in the following way:

$$\text{AUTOMATON2} \hat{=} \text{AUTOMATON1 } P [[\text{tea} \leftarrow \text{earlgrey}, \text{coffee} \leftarrow \text{marabacoffee}]]$$

This process is then equivalent to

$$\text{coin} \rightarrow (\text{marabacoffee} \rightarrow \text{AUTOMATON1} \square \text{earlgrey} \rightarrow \text{AUTOMATON1})$$

Renaming can for example be used to provide a common interface for sub-components, which allows them to use internally more sophisticated and clearer events.

2.2.4 Traces

In [7], the traces model is outlined by Roscoe, which is used for the specification checks.

A trace is the sequence of events that a process has visibly performed. This basically represents the history of a process. A trace of a process might be finite or infinite. The set $\text{traces}(P)$ contains all traces, that the process P might ever perform. For example, the set $\text{traces}(\text{STOP})$ is empty.

As another example, the traces of the process COFFEEAUTO1 is:

$$\text{traces}(\text{COFFEEAUTO1}) = \{ \langle \rangle, \langle \text{coin} \rangle, \langle \text{coin}, \text{coffee} \rangle \}$$

The set of traces for COFFEEAUTO2 is an example for an infinite trace set:

$$\text{traces}(\text{COFFEEAUTO2}) = \{ \langle \rangle, \langle \text{coin} \rangle, \langle \text{coin}, \text{coffee} \rangle, \langle \text{coin}, \text{coffee}, \text{coin} \rangle, \langle \text{coin}, \text{coffee}, \text{coin}, \text{coffee} \rangle \dots \}$$

For checking the trace behaviour of processes, the possibility of a trace refinement exists. As seen in [7], trace refinement is defined as:

$$P \sqsubseteq_T Q \equiv \text{traces}(Q) \subseteq \text{traces}(P)$$

In this definition, the process Q trace refines the process P, if and only if the traces of the process Q are a subset of the traces of P. In this context the process P is also called the *Specification* whereas the process Q is called the *Implementation*.

This specification is very useful in verifying the behaviour of processes by checking them with FDR.

Suppose a process POWERPLANT communicates the events *increase_temp*, *decrease_temp* and *explode* (the internal workings of the process may here be ignored). If one wants to verify, that the event *explode* can never happen, the following trace refinement should be verified:

$$\text{STOP} \sqsubseteq_T \text{POWERPLANT} \setminus (\Sigma \setminus \{\text{explode}\})$$

Every event that POWERPLANT can communicate is hidden, except for *explode*. FDR can then be used to check if indeed the set of all possible traces of this process is a subset of the traces of STOP, i.e. that POWERPLANT cannot communicate the event *explode*.

2.2.5 FDR as a Specification Checking Tool.

FDR (*Failure Divergence Refinement*) is a powerful tool, that can be used for model checking. It is the program, that is used to check the CSP models of security protocols.

It supports not only the above mentioned trace refinement check, but also failure and failure-divergence checks, which are other characteristics of processes, that can be used to compare the implementation of a specification. Furthermore, it can test, if a system deadlocks, goes into a livelock or is deterministic.

The CSP model of the security protocols are given to FDR in a machine-readable form called CSP_M . This notation is slightly different from the pure mathematical notation. For example, the process COPY has the original notation:

$$\text{COPY} \hat{=} \text{left?x} : T \rightarrow \text{right!x} \rightarrow \text{COPY}$$

In CSP_M it can be written as:

```
T = {1, 2, 3, 4}
channel left, right : T
COPY = left?x -> right.x -> COPY
```

FDR demands, that the used CSP model is finite and uses strict typing. For example, a process that counts the number of events that happened without any restriction cannot be checked. This system of working on finite models is one of the greatest concerns of checking security protocols, as the protocol might require the introduction of new values in every protocol run.

In [1] Broadfoot describes how FDR checks weather one *Implementation* process trace refines a *Specification* process. This is accomplished by doing a breath first search on the state space, which denotes the set of all possible traces of both processes. FDR tries to find a trace of the implementation process, that cannot be performed by the specification process. If FDR finds such a trace, the refinement check fails and the trace is presented to the user. This feature is especially useful, as the user can directly see, which course of events lead to the refinement to fail.

This is a problem for the new unbounded parallel model, whose implementation is presented in this dissertation, as a considerable part of the calculation of knowledge and the obtaining of values happens before the process actually runs, is hidden inside the system or just does not convey enough information what led to the failure.

As a breath first search is used, FDR will find the shortest trace error. This is very convenient, as the shortest trace does not contain events that are unnecessary for the refinement check to fail and might be easier to understand, because of its shortness.

2.3 The Bounded CSP Model for Checking Security Protocols

Security protocols can be modelled by the communication of the parallel running entities that are involved. It is therefore reasonable to use the CSP notation to model the workings of security protocols. This section describes the basics of the model, which is produced by Casper in the non data-independent mode. It will be referred to later as the *bounded model* or the *Casper model*.

The crucial feature of the bounded model is, that it makes specific assumptions about how many agents exist and how many times each of of them can run the protocol. Therefore, it is never possible to rule out, that an attack still exists, hence the bounded model is not complete. This is changed with the unbounded parallel model, which will be explained in Section 2.4.

The bounded CSP model makes the Dolev-Yao assumption [2]. This includes the assumption that the cryptographic methods of the security protocol are correct and not breakable and that the intruder in the network can eavesdrop, intercept, fake and duplicate any message within the cryptography constraint.

2.3.1 Data Types

Real world implementations of security protocols are usually composed of structured bit fields with attached interpretation rules. These messages need to be abstracted in CSP and it is necessary to keep information on how these messages are composed, for example to be able to conclude, what can be decrypted or encrypted.

In the Casper model, this is expressed in the *Encryption* data type, that holds all necessary structural information about messages. Atomic messages, as they are defined in Section 2.1, are represented as the set *Atoms*. Nonces, keys and the identities of agents are itself sets (*Nonce*, *Key*, *Agent*) and subsets of *Atoms*. Other atomic data types in messages are represented in the same way, i.e. further sets as subsets of *Atoms* are introduced.

All operations on the atomic type are defined as recursive data types. This gives the *Encryption* data type the following definition:

$$\text{Encryption} := \text{ATOM.Atom} \mid \text{ENCRYPT.Encryption.Encryption} \mid \text{Sq.Encryption}^*$$

The above data type can be expanded, as more and different recursive definitions of messages can be added, and indeed the implementation of the unbounded parallel model will require such an extension (see Section 2.4.3).

An instance of Message 3b. of the Yahalom protocol would be constructed using the above definition as:

$$\text{Encrypt.ServerKey}(\text{Bob}).(\text{Sq.}\langle \text{ATOM.Alice}, \text{ATOM.Kab} \rangle)$$

The future notation of messages will be as follows: $\{m\}_k$ for a message that is encrypted with the key k , $\langle m1, m2, m3 \rangle$ or just $m1, m2, m3$ for the sequence $\text{Sq.}\langle m1, m2, m3 \rangle$. Furthermore, atoms will directly be named Na or Kab or $Alice$, instead of ATOM.Na and so forth.

2.3.2 Trustworthy Agents

The involved parties of security protocols are so called agents. They assume the roles of the protocols, which were previously just called a or b or s . Often, security protocols involve two roles: the initiator and the responder. Sometimes, an additional server supports the initiator and responder by generating keys or certifying identities.

The roles of the security protocol are modelled in CSP directly as processes and the agents are then defined as compositions of the roles, that they assume.

The following events are important:

- send.a.b.m denotes a message that is sent from agent a to agent b with the content m
- receive.a.b.m denotes, that agent b received the message with the content m from a
- env.b is a message, that tells the receiver of the message to run the protocol with agent b (this is the first message for the process in the system and not part of the actual protocol)

The first two are used for the communication in the network while the third event helps in modelling the process.

The intruder will be able to create these messages freely within the cryptography constraint, so that the identities of sender and receiver might be forged.

The protocol must be carefully translated into a series of receive and send events, whereas every process has its separate series of events to follow.

For example, the process a (the Initiator) in the Yahalom protocol only engages in messages 1, 3a and 4. His view on the course of actions is therefore:

- Message 1 : send the nonce na to b

- Message 3a : receive from an agent, who gets the identity s , a message in the form $\{b, kab, na, nb\}_{\{ServerKey(a)\}}$; the message is only accepted, if the message is correctly encrypted and the atoms b and na are equal to the previously used ones
- Message 4 : send the message $\{nb\}_{\{kab\}}$ to b

The process that simulates the initiator behaviour can then defined in CSP as:

$$\begin{aligned} \text{Initiator}(a, na) &\hat{=} \\ &\text{env?}b : \text{Agent} \rightarrow \\ &\text{send.a.b.na} \rightarrow \\ &\square_{kab \in Key, nb \in Nonce, s \in Server} \left(\begin{array}{l} \text{receive.s.a.}\{a, kab, na, nb\}_{\{ServerKey(a)\}} \rightarrow \\ \text{send.a.b.}\{nb\}_{\{kab\}} \rightarrow \text{Session}(a, b, na, nb, kab) \end{array} \right) \end{aligned}$$

The composition of the events ensures, that the cryptography is not broken and the variables match during the protocol run.

Other processes – in the case of the Yahalom protocol the responder and the server – are build in a similar way.

One agent can impersonate several of the roles. Suppose agent Alice impersonates the roles of Initiator and Responder, then all roles are composed using the interleaved parallel operator (modified from [7]):

$$\begin{aligned} \text{Agent_Alice} &\hat{=} \\ &(\parallel_{na \in Nonce_IAlice} \text{Initiator}(Alice, na)) \\ &\parallel \\ &(\parallel_{nb \in Nonce_RAlice} \text{Responder}(Alice, nb)) \end{aligned}$$

Finally, the trustworthy agents of the network, Alice and Bob, can be put together as:

$$\text{Network} \hat{=} \text{Agent_Alice} \parallel \text{Agent_Bob}$$

2.3.3 The Intruder

In the network the intruder is modelled as a separate process. As there are no general rules, how to expose specific security attacks, every possible combination of messages and security runs need to be performed.

Based in the Dolev-Yao model, the intruder can overhear messages and learn from them, block messages and create messages with the knowledge that the intruder already possesses.

For this purpose, the intruder can pretend to be one of the other agents in the network. Usually, the intruder also has his own identity as one of the agents in the network, including cryptographic keys of this agent.

The intruder starts with initial knowledge IK_0 . By overhearing messages and with the help of a deduction system, more knowledge can be gained and inferred.

Deductions are formed as pairs (X, f) , with the meaning: If the intruder has all knowledge of X , f can be inferred.

$$X \vdash f$$

All possible deductions are subject to the cryptography constraint, the following deductions are possible:

- Encrypting messages with known keys: $\{k, m\} \vdash \{m\}_k$, for all keys k and messages m
- Decrypting messages with known keys: $\{m\}_k, k^{-1} \vdash m$, for all keys k and messages m
- Building sequences: $\{m_1, \dots, m_n\} \vdash Sq. \langle m_1, \dots, m_n \rangle$, for all sets of messages $\{m_1, \dots, m_n\}$
- Decomposing sequences: $Sq. \langle m_1, \dots, m_n \rangle \vdash \{m_1, \dots, m_n\}$, for all sequences $\langle m_1, \dots, m_n \rangle$

These deductions are combined in the set *CoreDeductions*. The set of deductions that is finally used is the set *BaseDeductions*, which is equal to *CoreDeductions* for the bounded model. *BaseDeductions* will contain more deductions in the unbounded model.

Usually, the following CSP notation is given for the behaviour of the intruder ([7]):

$$\begin{aligned} \text{Intruder}(X) &\hat{=} \text{hear?m : messages} \rightarrow \text{Intruder}(\text{close}(X \cup \{m\})) \\ &\quad \square \text{say?m : } X \cap \text{messages} \rightarrow \text{Intruder}(X) \\ &\quad \square \text{leak?f : } X \rightarrow \text{Intruder}(X) \end{aligned}$$

The function $\text{close}(X)$ applies the deductions on the set X until all possible facts are generated.

As FDR is only able to test finite systems, the above intruder cannot be modelled exactly like this.

Instead, before the system is checked, the following sets are calculated:

- IIK_1 contains all knowledge, that can be inferred from the intruder's initial knowledge IIK_0 using *BaseDeductions*
- *Deductions*, the set of all deductions that were not inferred yet

- *KnowableFact*, the set of all facts, that are not known yet, based on the conclusions of all deductions that were not inferred yet plus all observable protocol messages
- The set *LearnableFact* contains every fact, that might be learned through the running of the protocol.

These sets are calculated with the function $\text{Close}_{\text{Fact}_1}^{\text{BaseDeductions}}(X)$, where Fact_1 is the set of all atoms, structures and sub-structures as well as sent-facts, that might ever occur and X is the set of premise facts, from which all inferences are made.

Furthermore, the set *LearnableFact* is the set *KnowableFact* minus the knowledge IIK_1 :

$$\begin{aligned} (\text{IIK}_1, \text{Deductions}, \text{KnowableFact}) &= \text{Close}_{\text{Fact}_1}^{\text{BaseDeductions}}(\text{IIK}_0) \\ \text{LearnableFact} &= \text{KnowableFact} \setminus \text{IIK}_1 \end{aligned}$$

These sets are calculated exactly once within the bounded model.

One process is created for every fact from the set *LearnableFact*. This process is initially in an ignorant state. If he is able to hear or infer the fact, he switches to a knowing state, in which he is able to communicate the fact. It is therefore not necessary to apply the Close-function in this situation, as the process knows, which other facts need to be in the known state, so that it can be inferred.

For this purpose, the internal fact-processes communicate *hear* and *infer* events. The *infer* event in these fact-processes is hidden in the intruder. The deduction function in the intruder is monotonic. Performing one inference will never prevent the inference of another fact, so that the order of inferring does not matter. FDR usually considers all possible orders, which is unnecessary in this case. For this reason, the function *chase()* was introduced. The process *chase(P)* behaves like P , just that hidden events are performed until no more hidden events are available, without caring for the order. If two hidden events are available, one of them is chosen randomly. Thus, the chase function is applied to the inference-system, so that the deductions are applied in the fastest way possible, without exploring the state space unnecessarily.

Another single process is used to communicate facts from the set IIK_1 .

This knowledge about the composition of the intruder is important, because more learnable facts make more processes in the system necessary, which in turn slows down the compilation of the CSP script by FDR.

The process, that uses the above described fact-processes, will be called *Intruder'*.

2.3.4 Connecting the Components

The complete system consists of the network of agents (the process *Network*), connected with the intruder (the process *Intruder'*).

The process *Network* may contain several initiators, responders and servers with a number of nonces and keys. The events of the intruder are renamed, so that they conform with the internal events of the network.

The complete system is then:

$$\begin{aligned} \text{System} &\hat{=} \\ &\text{Network} \\ &\quad || \\ &\text{Intruder}'[[\text{hear}.m \leftarrow \text{send}.a.b.m, \text{say}.m \leftarrow \text{receive}.a.b.m| \\ &\quad a \in \text{Agent}, b \in \text{Agent}, m \in \text{Messages}]] \end{aligned}$$

The intruder will be given one identity, that he can use to run the protocol with, which is sufficient according to [3].

The core problem of the bounded model is the incompleteness. The number of agents and values in the system will influence the attacks, that can be found. It is not possible to rule out, that more agents could not disclose more security flaws.

2.3.5 Specifying Protocol Goals

The goals of security protocols are checked by running a trace refinement check as outlined in Section 2.2.5. This is done by adding several events to the system, that express the beliefs of the agents in the system. The system is then tested against a specification process, that can only communicate the correct order of events.

For security protocols, this is done as:

$$\text{SPEC} \sqsubseteq_T \text{SYSTEM} [[R]] \setminus (\Sigma \setminus \alpha(\text{SPEC}))$$

SPEC is the specification process and SYSTEM is the previously outlined system of trustworthy agents and the intruder. All events of the system are hidden, except of the specification events, which are denoted as $\alpha(\text{SPEC})$. Furthermore, some of the events of the system are renamed to the above mentioned signalling events according to the renaming relation R.

Secrecy

For checking secrecy, a new event is introduced:

$$\text{Claim_Secret}.a.b.s$$

This event expresses, that *a* believes at the point when this event can be communicated, that only the honest agent *b* (besides *a*) knows the secret value *s*. If the intruder is able to

know the value s and perform the event $\text{leak}.s$, the protocol does not guarantee the secrecy of s .

The original system is changed by renaming the last message sent by the a to the above mentioned $\text{Claim_Secret}.a.b.s$ for the specification check.

According to [8], the specification process is:

$$\begin{aligned} \text{Secret_Spec}_0(s) &\hat{=} \\ &\text{Claim_Secret}?a?b.s \rightarrow (\text{if not honest}(b) \text{ then } \text{Secret_Spec}_0(s) \text{ else } \text{Secret_Spec}_1(s)) \\ &\square \text{leak}.s \rightarrow \text{Secret_Spec}_0(s) \\ \text{Secret_Spec}_1(s) &\hat{=} \\ &\text{Claim_Secret}?a?b.s \rightarrow \text{Secret_Spec}_1(s) \end{aligned}$$

This expresses, that $\text{leak}.s$ must not occur, when a claims s to be a secret in a session with the honest agent b .

Authentication

Authentication can be defined in several ways and there are different ideas of what constitutes a correct authentication. The following descriptions shall give an introduction into specifying and testing authentication goals.

For the authentication specification, the events *Running* and *Commit* are introduced. The event *Commit* is communicated by honest agents when the protocol run is completed. *Running* on the other hand is communicated by honest agents before the last sent event to express the belief, that the agent is in a session with the other honest agent.

The event $\text{Commit}.b.a$ expresses, that b has completed the protocol run with whom he believes to be agent a . The event $\text{Running}.a.b$ means, that a is running the protocol apparently with b .

The security protocol ensures authentication if after a $\text{Running}.a.b$ always follows a $\text{Commit}.b.a$ event.

Furthermore, these events can be associated with variables to express that a and b believe, that they agree on the same values for this variable. The above authentication notation is then extended in the following way: For an agreement on variable v , on the event $\text{Running}.a.b.v$ must always follow a $\text{Commit}.b.a.v$ event.

The very specific notion of injective authentication, which requires that there must not be more completed protocol runs than started, is then specified as:

$$\text{Auth_Spec}_0(a,b,ds_a,ds_b) \hat{=} \\ \text{Running}.a.b.ds_a,ds_b \rightarrow (\text{Commit}.b.a.ds_a,ds_b \rightarrow \text{STOP} ||| \text{Auth_Spec}_0(a,b,ds_a,ds_b))$$

In the above example, ds_a and ds_b are variables of a and b on which they both have to agree.

This specification itself cannot be tested in FDR, as the recursive parallelism introduces infinitely many processes. Hence, one has to specify a limited number of parallel processes in the following form:

$$\text{Auth_Spec}_0(a,b,ds_a,ds_b) \hat{=} \\ \text{Running}.a.b.ds_a,ds_b \rightarrow \text{Commit}.b.a.ds_a,ds_b \rightarrow \text{STOP}$$

$$\text{Auth_Spec}_1(a,b,ds_a,ds_b) \hat{=} \\ \text{Auth_Spec}_0(a,b,ds_a,ds_b) ||| \dots ||| \text{Auth_Spec}_0(a,b,ds_a,ds_b)$$

2.4 The Unbounded Parallel model

FDR and the bounded Casper model have been proven to be very useful in finding security attacks in protocols. Nonetheless, the model cannot be used to prove the absence of security flaws under the Dolev-Yao assumption, as it is bounded in the number of instances of the security protocol, that can run at the same time.

This was solved in the so called Data Independence model (not to be confused with the notion of data independence itself), outlined for example in [1]. It allows models, in which agents can run an unbounded number of sequential runs of the protocol. Unfortunately, that model has been shown to take very long to be checked using FDR, making it infeasible to check very large and complex models.

Kleiner and Roscoe introduced in [4] a new, unbounded model, which has been proven to be both complete and much more efficient than the bounded and the Data Independence model. The model was extended and the most current theoretical work on the model can be found in [3].

The model is able to reveal security and authentication attacks on static protocols, which perform a fixed number of steps with a bounded number of messages that are sent or received. Additionally, no state is carried forward to the next run of the protocol.

This happens while simulating an infinite number of parallel instances of the protocol running at the same time. For this reason, the model will be referred to as the *unbounded parallel model*. If it does not find an attack on a protocol, then no attack under the Dolev-Yao assumption exists.

The unbounded parallel model is based on the same model that was described in the previous sections. This model is extended by internalised agents, using a collapsing function on values, and changing the way the refinement checks are modelled.

Furthermore, the internalisation of the agents allows to improve the compilation of CSP scripts by FDR, as it is possible to decrease the number of learnable facts and therefore the number of processes in the system.

Within this section, the practical implications and methods of the unbounded parallel model are explained. Theoretical explanations and proofs can be found in [4] and [3].

The technique of internalising agents was already used in the Data Independence model and is therefore not new in the unbounded parallel model. Internalised agents work as an oracle within the intruder. They are not modelled as processes, but as deductions, that extend the deduction set of the intruder. This creates the effect of being able to run infinitely many parallel instances of the protocol.

Internalised agents are usually divided into two groups: Agents that generate new values and agents that do not. The first group of agents can be easily modelled, while the latter were usually seen as more difficult. Past models used "manager processes" with value-collections to accomplish correct generation deductions. This has been shown to be inefficient.

The new model does not demand that external agents are changed in any way. But some improvements effect them, too.

2.4.1 Data Independence

Within process algebras, values like nonces and keys are usually treated as data independent, because all operations on them are either comparisons for equality or polymorphic constructors like symbolic encryption or tupling.

In the unbounded model, a collapsing function ϕ is applied on all data independent types T of a process $P(T)$ and it is proven, that the behaviour of $P(T)$ is sufficiently similar to the behaviour of $P(\phi(T))$, so that it is possible to expose any attack. The collapsing function ϕ is important, as it will be used to generate values correctly within the deduction system.

The behaviour of a process is defined by its traces, so that the following equation holds:

$$\text{traces}(P(\phi(T))) = \{ \phi(t) \mid t \in \text{traces}(P(t)) \}$$

According to [4] this is automatically true, when ϕ is an injective function. When ϕ is non-injective, more conditions need to be fulfilled, namely the *PosConjEqT'_C* and the *Positive Deductive System*. They state that "no external agent ever requires an inequality (except with the members of the set C of constants) to make progress", as well as that the intruder does not depend on inequalities within the deductions.

These conditions hold for the protocols that are examined in this dissertation.

2.4.2 The Collapsing Function

Suppose one wants to model the Yahalom Public Key protocol (listed in Section 2.1) with the honest agents Alice and Bob as well as the intruder *Ivo*. In the protocol, every honest agent generates one fresh nonce per run. The following collapsing function was proposed in [4] and extended in [3]:

$$\phi(X) = \begin{cases} X & \text{if } X \in \text{Agent is the identity of one of the external agents;} \\ \text{Alice} & \text{if } X \in \text{Agent is the identity of one of the honest internal agents;} \\ \text{Ivo} & \text{if } X \in \text{Agent is any other agent's identity;} \\ X & \text{if } X \in \text{Nonce is generated by one of the external agents;} \\ N_s & \text{if } X \in \text{Nonce is secret and is generated by an honest internal agent} \\ & \text{in a session with an honest agent;} \\ N_p & \text{if } X \in \text{Nonce and } X \text{ is generated by an honest internal agent in a} \\ & \text{session with the intruder;} \\ N_p & \text{if } X \in \text{Nonce is not secret;} \\ X & \text{if } X \in \text{Nonce is generated by an honest internal agent in a session} \\ & \text{with with an honest agent and is a additional nonce to the above} \\ & \text{used } N_s \text{ and } N_p \text{ values} \end{cases}$$

It is easy to see that the identities of *Alice*, *Bob*, *Ivo* and external nonces Na and Nb are mapped to themselves. Only the nonces that are generated internally are mapped to special values. Whenever the intruder is involved in the session or the nonce can be seen by people that are not involved in the protocol (i.e. the nonce is not secret), the nonce N_p is used. In the case of sessions involving only honest agents, a generated nonce gets the value N_s .

Of course, a secrecy decision needs to be done, to distinguish apparently secret nonces from apparently public nonces. In the case of the Yahalom protocol, the nonce na is public, as it is sent in message 1 unencrypted, while the nonce nb is secret.

The last mapping is necessary to introduce additional values into the system, that are necessary for authentication. For example: if one wants to use four values for a secret variable, then one of these values is mapped to N_s and the additional three are mapped to themselves. These values are, as explained in [3] only necessary for the authentication check for variables, that are generated by the initiator.

The collapsing function can be extended to all generated variables, that follow the assumptions, that were outlined in Section 2.4.1, using the values T_s and T_p for the type T. This applies in the Yahalom example to the key kab , that is mapped to Kab_s and Kab_p , following the above outlined rules.

Already here can be seen, that no changes to the external agents need to be made, as the

collapsing function does not change their values. The collapsing function only applies to values that were generated by internal agents.

The above outlined collapsing function means, that the system is reduced to a finite one and does not need an infinite pool of values. For message 3b of the Yahalom protocol, all messages that the intruder can generate with the internal initiator deductions are:

$$\{Alice, Kab_s\}_{ServerKey(Alice)}, \{Alice, Kab_s\}_{ServerKey(Bob)}, \{Alice, Kab_p\}_{ServerKey(Ivo)}, \{Ivo, Kab_p\}_{ServerKey(Ivo)}, \{Ivo, Kab_p\}_{ServerKey(Bob)}$$

2.4.3 Internalising Agents

The internal agents are modelled as deductions in the inference system of the intruder.

Suppose, the Yahalom protocol is internalised, using one internal initiator, one internal responder and one internal server. Message 1 of the Yahalom protocol gets mapped to Np . Message 2 as response can then be modelled as a deduction in a generalised way. Note that this version is not the final used version, as it is unrestricted.

For sessions with honest agents, the following deduction is used:

$$\forall b \in Honest, na \in Nonce \bullet na \vdash \{Alice, na, Ns\}_{ServerKey(b)}$$

For sessions with dishonest agents, the following deduction is used:

$$\forall b \in Honest, na \in Nonce \bullet na \vdash \{Ivo, na, Np\}_{ServerKey(b)}$$

Internal deductions need to be modelled very carefully, to avoid surplus behaviour. For example, an internalised agent should not be able to deduce message 4 of the Yahalom protocol, without that message 1 ever happened.

Furthermore, the internalised agent needs to have some information about previously generated variable to make sure, that the values match. This is, in the Yahalom protocol, important for messages 3a and 3b, where the variable kab in both messages must match. Therefore, there must be some information about this value in the deduction.

In this model these restrictions are implemented by adding a new construct to the *Encryption* type, which includes the definition of two new types:

$$\begin{aligned} \text{Encryption} &:= \text{ATOM.Atom} \mid \text{ENCRYPT.Encryption.Encryption} \mid \\ &\quad \text{Sq.Encryption}^* \mid \text{Sent.Message.Tag} \\ \text{Message} &:= \text{Encryption} \\ \text{Tag} &:= \text{Sq.Encryption}^* \end{aligned}$$

Message is equal to type *Encryption* and *Tag* to type *Sq.Encryption**. The new *Sent* construct is abbreviated as *Sent(m,t)*. A sent-fact has to be read in the following way:

At the point in time the message m was sent, the variables of t were instantiated in the sender.

For example, by the time message 1 of the Yahalom protocol is sent, the variables a , b and na are instantiated. The following sent-facts would then exist (under the same assumption as the previous non-generalised deductions):

$$\text{Sent}(\text{Np}, \langle \text{Alice}, \text{Bob}, \text{Np} \rangle), \text{Sent}(\text{Np}, \langle \text{Alice}, \text{Ivo}, \text{Np} \rangle)$$

These sent-facts then have to be used as premises for deductions of messages, that are preceded by other messages sent by the same sender. For example, message 4 of the Yahalom protocol would need as premise the previous sent message 1 and the previous received message 3b:

$$\forall a, b \in \text{Agent}, \quad \bullet \left\{ \begin{array}{l} \text{Sent}(na, \langle a, b, na \rangle) \\ \{b, kab, na, nb\}_{\text{ServerKey}(a)} \end{array} \right\} \vdash \text{Sent}(\{nb\}_{kab}, \langle a, b, na, kab, nb \rangle)$$

This example shows the restriction on agent a of the protocol. Message 1 needs to be sent to be able to send message 4, as the intruder cannot deduce sent-facts without using the internalisation-deductions. Furthermore, all values that were previously used are remembered in the Tag-part of the sent-fact, so that they can be used in any later part of the protocol.

Therefore, a final version of the deduction for message 2 can be built.

For sessions with honest agents, the following deduction is used:

$$\forall b \in \text{Honest}, na \in \text{Nonce} \bullet na \vdash \text{Sent}(\{ \text{Alice}, na, \text{Ns} \}_{\text{ServerKey}(b)}, \langle b, \text{Sam}, na, \text{Alice}, \text{Ns} \rangle)$$

For sessions with dishonest agents, the following deduction is used:

$$\forall b \in \text{Honest}, na \in \text{Nonce} \bullet na \vdash \text{Sent}(\{ \text{Ivo}, na, \text{Np} \}_{\text{ServerKey}(b)}, \langle b, \text{Sam}, na, \text{Ivo}, \text{Np} \rangle)$$

Since the intruder needs to know the message within the every sent-fact for later reuse, the following general deductions need to be included:

$$\text{SentDeduction} \hat{=} \forall m \in \text{Messages}, t \in \text{Tags} \bullet \text{Sent}(m, t) \vdash m$$

2.4.4 Secrecy Analysis

Secrecy analysis has greatly changed in the unbounded parallel model. According to [3], no external agents are necessary anymore. Hence, secrecy analysis can be done entirely using the internal deductions of the intruder with one internal agent for every role.

Secrecy is then ensured by the protocol, if the intruder is not able to *leak* one of the internal T_s values, as they are supposed to stay secret in the run between honest agents.

Therefore, the trace refinement check is:

$$\text{STOP} \sqsubseteq_T \text{SYSTEM} \setminus (\Sigma \setminus \{\text{leak.s} \mid s \in \text{InternalSecretValues}\})$$

The check of this specification will succeed, if the intruder is not able to leak a secret variable and it will fail otherwise. Therefore, the mapping of variables to the public T_p value or the secret T_s value is crucial in defining, which variables must never be leaked.

2.4.5 Authentication Analysis

For the authentication checking to succeed, some changes had to be done. Foremost, the system still needs to be able to communicate the *Running* event. As some sessions are run between internal and external agents, it is not enough to rename internal send-events to the corresponding *Running* events. Instead, the inferences of the corresponding messages need to be renamed to the appropriate running event.

For example, for the Yahalom protocol the renaming was as follows:

$$\text{SYSTEM}[[\text{send.a.b.}(\text{Msg4}, \{\text{nb}\}_{\text{kab}}, \langle \text{na}, \text{nb} \rangle) \leftarrow \text{Running.a.b.na.nb}]]$$

The corresponding renaming of the infer-event within the intruder is then (INTRUDER_00 is the intruder before the infer events are hidden):

$$\text{INTRUDER_00} [[\text{infer.a.b.Sent}(\{\text{nb}\}_{\text{kab}}, \langle \text{a}, \text{b}, \text{na}, \text{kab}, \text{nb} \rangle) \leftarrow \text{Running.a.b.na.nb}]]$$

It is not necessary to rename internal events to *Commit* events, as outlined in [3].

One great problem of the newly introduced renaming are inferences, that were already done by the initial calculation of $\text{Close}(\text{IIK}_0)$. As specific events would then already be known, the corresponding *infer* event would never occur. This problem can be solved by preventing the Close function to calculate facts, that should be renamed later. Another method would be to actual calculate these facts, but keeping track of the necessary *Running* for later communication within the system.

3 Casper implementation of the Unbound Parallel model

Casper currently implements a sophisticated version of the bounded version explained in Section 2.3 as well as the Data Independence model as presented in [1]. To implement the unbounded parallel model, several changes to the Haskell source code of Casper had to be made.

First, the user must get the means to use the new model. That means, that the Casper input notation needs to be extended to accommodate the changes of the unbounded parallel model. Furthermore, agents need to be internalised by using the deduction system of the intruder.

3.1 Changes to the Casper Scripts

The new model should be utilizable by the user in a similar way to the bounded model and the data independence model, without the need for too much detailed knowledge. Therefore, minimal changes to the way the security protocol is defined in the Casper script were made. Some of these changes are similar to the changes that were made for the Data Independence model (as discussed in [1], section 4.3.1). This section will discuss all changes and how the user can model security protocols using these extensions.

All examples in this section are take from the Yahalom protocol: the complete script is available in appendix A.2

3.1.1 The `#Processes` Section

The user must indicate which variables are generated newly for every protocol run. These variables will be treated by the collapsing function outlined in Section 2.4.2. In the `#Processes` section, these variables are still part of the argument list of the process, but they are separately marked as being generated by the keyword `generates`.

The following is an example from the Yahalom protocol:

```
9 #Processes
10 INITIATOR(a,na) knows ServerKey(a) generates na
11 RESPONDER(b,s,nb) knows ServerKey(b) generates nb
12 SERVER(s,kab) knows ServerKey generates kab
```

Here, the variables `na`, `nb`, and `kab` are data independent and are therefore collapsed. These variables are provided by the user in the `#System` section.

3.1.2 The #Specification Section

The secrecy specifications in the #**Specification** did not change. But, if the unbounded parallel model is produced and one **Secret**-line in this section exists, an additional CSP trace specification is inserted automatically. As described in Section 2.4.4, this specification in this produced model will try to check, if the system is able to leak variables with the status *InternalUnknown* (see below), so that the user has to choose variables that have to be secret by using values with this status in the #**System** section.

3.1.3 The #Actual variables Section

In the #**Actual variables** section, variables get new statuses assigned. These statuses are *Internal*, *External*, *InternalUnknown* and *InternalKnown*. These are the only values to be used for instantiating generated variables. Advice, how many of these values are to be declared and for which kind of checking will follow in Section 3.1.6.

The following is again taken from the Yahalom protocol:

```
26 #Actual variables
27 Alice, Ivo : Agent
28 Sam : Server
29
30 Kabp : SessionKey(InternalKnown)
31 Kabs : SessionKey(InternalUnknown)
32
33 Nb : Nonce(External)
34 Np : Nonce(InternalKnown)
35 Ns : Nonce(InternalUnknown)
36 Na1i : Nonce(Internal)
```

Values with the status *External* are to be used only in the definition of external agents.

The other statuses must only be used for values, that instantiate generated variables of internal agents. Every process must use either only values with the status *Internal* or only values with one of the statuses *InternalUnknown* and *InternalKnown*.

The status *InternalKnown* must be used for values that instantiate variables, that are apparently not secret, i.e. someone who does not take part in the protocol run might be able to get access to it, this corresponds to the T_p value. On the other hand, the status *InternalUnknown* must be used for values that instantiate variables that are apparently secret to outsiders, therefore constituting the T_s value.

3.1.4 The #System Section

There are several changes in the `#System` section. First, the keyword **internal** is introduced. Any process, that is marked as internal, is not modelled as real process in the CSP script, instead the process' behaviour will be simulated by internal deductions. Any process without this keyword is external.

```
43 #System
44 INITIATOR(Alice, Np) internal
45 INITIATOR(Alice, Nali) internal
46
47 RESPONDER(Alice, Sam, Ns) internal
48 RESPONDER(Alice, Sam, Nb)
49
50 SERVER(Sam, Kabs) internal
51
52 DecideSecrecy = False
53 UnboundOptimize = True
```

There are two new options introduced. The option *UnboundOptimize* is used to indicate, that the user wants to enable the optimisation of the compilation procedure. This procedure reduces the time to check the protocol considerably (benchmark results can be found in Chapter 5). Nothing else changes for the user. The default value for this option is *false*.

[4] establishes the need of a secrecy analysis of the values, which is necessary for the collapsing function. This secrecy analysis can either be done by the user or automatically by the system. The option *DecideSecrecy* is used to activate the automatic secrecy decision procedure. The default value for this option is *false*.

The number of necessary agents and their configuration for specific situations is established in [3] and depends on the type of check that is done. This section will not explain them (see Section 3.1.6), but the general rules for a possible configurations.

Generally, one must configure one internal agent for every process. These agents must generate values that have the status *InternalKnown* or *InternalUnknown*. Which status to choose depends on the apparent secrecy of the variable.

Additional internal agents that generate values with the *Internal* status are used to bring more values into the system, which is necessary for some authentication checks.

The internal agent, that generates *InternalUnknown* or *InternalKnown* values, does not need to be defined, if the user enables *DecideSecrecy* and defines at least one of the agents, that generates values with the status *Internal*. The absent, but necessary internal agent is then added automatically.

External agents can be added to the system as necessary.

Table 3.1 gives an overview of what constitutes a possible configuration.

Type of agent	DecideSecrecy = False	DecideSecrecy = True
Internal agent with values with the status <i>InternalUnknown</i> or <i>InternalKnown</i>	one for every process	none (automatically added)
Internal agent with values with the status <i>Internal</i>	unrestricted	at least one for every process
External agent with values with the status <i>External</i>	unrestricted	unrestricted

Tab. 3.1: Configuration of internal and external agents

This principle is illustrated in the above excerpt from the Yahalom description. The script is used to check the absence of flaws in the non-injective authentication. For this, two internal initiators, one internal responder and server and one external responder are used.

In the Yahalom protocol, the variable na is not encrypted in message 1, so that the value Np must be used for the initiator-configuration. The variable nb is encrypted in every run and the concerning messages are not decryptable by an outsider, so that the value Ns is used for the responder-configuration.

3.1.5 The `#Intruder Information` section

Finally, the `#Intruder Information` section was changed. The new option **UnboundParallel** is used to explicitly enable the output of a CSP script, that uses the unbounded parallel model. This option is *false* by default.

Furthermore, the intruder's initial knowledge must contain every value with the status *InternalKnown*.

```

57 #Intruder Information
58 Intruder = Ivo
59 IntruderKnowledge = Alice, Ivo, Sam, Np, Kabp, ServerKey(Ivo)
60 UnboundParallel = True

```

3.1.6 Correct Configuration for Different Checks

In Section 3.1.4, the possible number and configuration of agents was established. This was done to give as much choice as possible to the user, who might test different configurations and number of nonces and keys.

In [3] however, Eldar Kleiner established the number of external and internal initiators and responders and their respective nonces and keys that are to be used in the unbounded

parallel model for different security properties. It was shown, that if no attack is found with the defined configuration, then no attack upon the protocol (under the Dolev-Yao assumption) exists.

According to Corollary 2.5.3 in [3], secrecy analysis should be done using one internal agent as initiator, one internal agent as responder and if applicable one internal agent as server.

All internal agents have to use values, that have the status *InternalKnown* or *InternalUnknown*, according to their apparent secrecy. If FDR is able to find a trace, that leaks on of the values with the status *InternalUnknown*, a secrecy attack is found. The user therefore configures by his choice of values, which variables he thinks are secure and have to remain secret. This also means, that the option *DecideSecrecy* must be disabled.

For the non-injective authentication, in accordance with proposition 2.5.4 of [3], two internal initiators, one internal, one external responder and one internal server, if applicable, have to be used. One of the internal agents for every process must, as established in 3.1 use only values that have either the status *InternalKnown* or *InternalUnknown*, according to their apparent secrecy.

For the injective authentication, one extra external responder has to be added, in accordance with proposition 2.5.12 of [3].

The option *UnboundOptimize* will improve the speed of the check considerably and should be enabled for both the injective and non-injective authentication.

The option *DecideSecrecy* can be used for authentication checks by configuring one internal agent for every process with values with the status *Internal* for generated variables. This will, of course, make some more values necessary, which will influence the speed negatively. The reason for this drawback is the internal working of the implementation of the unbounded parallel model: The addition of this automatic internal agent is done by mapping an already existing one to the correct values, this also guarantees correct values for variables, that are not generated.

3.2 Construction of the Internal Agents

For internalising agents within the intruder, they must be modelled as deductions. The values used in the system are subject to a collapsing function, effectively mapping the possibly infinite system to a finite one.

The user will supply a number of arguments, that are used for the internal agents' protocol runs with other honest agents. These arguments are similar to the assignments that are used for traditional external agents.

3.2.1 Implementing the Collapsing Function

The collapsing function used in the unbounded parallel model was explained in Section 2.4.2.

It can be easily seen, that the majority of the details of the function either do not make changes necessary or can be already followed by the user when creating the Casper input.

The following mappings happen in this way:

$\phi(X) =$	Condition	How it is implemented
X	$X \in \text{Agent}$ is the identity of one of the external agents;	Generations of external processes are not changed, the values within the deduction system are never mapped
Alice	$X \in \text{Agent}$ is the identity of one of the honest internal agents;	The user configures only one identity for the internal agents
Ivo	$X \in \text{Agent}$ is any other agent's identity;	The user gives only one identity to the intruder
X	$X \in \text{Value}$ is generated by one of the external agents;	Generations of external processes are not changed, the values within the deduction system are never mapped

These rules are not strict, the user is allowed to give more identities to the internal agents, for example. This will not change the validity of the model as it is still finite. But as a consequence of introducing more identities for internal agent the performance will suffer, it is therefore discouraged to use more identities for internal agents.

The remaining mappings of the collapsing function are:

No	$\phi(X) =$	Condition
1	T_s	$X \in T$ is secret and is generated by an honest internal agent in a session with an honest agent;
2	T_p	$X \in T$ and X is generated by an honest internal agent in a session with the intruder;
3	T_p	$X \in T$ is not secret;
4	X	$X \in T$ is generated by an honest internal agent in a session with with an honest agent and is a additional nonce to the above used N_s and N_p values

The idea of the implementation is, to make two sets of deductions within the intruder: Those involving only honest agents (rules 1,3 and 4 apply) and those that involve the intruder (rules 2 and 3). The values for deductions involving only honest agents are supplied by the user. The values for sessions involving the intruder are assigned automatically, as this is simply a mapping to the value T_p .

The following Initiator definition in the #Processes section is from an imagined protocol:

INITIATOR(a, s, na1, na2) knows ServerKey(a) generates na1, na2

The following values are supplied by the user, assuming *na1* is public and *na2* is secret:

```
INITIATOR(Alice, Sam, Np, Ns)  internal
INITIATOR(Alice, Sam, Na1i, Na2i)  internal
```

Therefore, the user assigns the generated variable *na1* the mapping *Np* (rule 3) and *Na1i* (rule 4) and the variable *na2* gets the values *Ns* (rule 1) and *Na2i* (rule 4).

If the user does not want to decide the secrecy of variables himself (rules 1 and 3), he can supply assignments with the mappings of rule 4 and the system will automatically add the correct necessary values (see 3.3).

On the other hand, for sessions involving the intruder, all generated variables need to get the variable T_p assigned. This is done in this implementation by taking the above user supplied variable assignments and substituting all generated variables with the appropriate T_p value.

For the above mentioned example, the initiator `INITIATOR(Alice, Sam, Np, Np)` will be used for communicating with the intruder.

3.2.2 Building the Deductions

General Concept

For the intruder being able to simulate the internal agents by deductions, every legal combination of deductions need to be included in the intruder's deduction system.

The user has a great flexibility in how to configure internal agents. Furthermore, the outlined application of the collapsing function requires some sophistication in how the deductions are composed.

Therefore, the implementation of the deductions was made in a way, that allows the greatest flexibility.

Firstly, the parameterised deductions are generated from the protocol description. These parameterised deductions do not contain yet any assignment to the variables involved in the protocol run, instead they can be used as functions for any variable assignment.

For the Yahalom protocol, all involved variables are *a*, *na*, *b*, *s*, *nb*, and *kab*.

The function containing all deductions could be

$$\text{parameterdeductions}(a, na, b, s, nb, kab) = \{ \dots \}$$

The set of all deductions concerning internalised agents would then be:

$$\text{AllDeductions} = \{ \text{parameterdeductions}(a, na, b, s, nb, kab) \mid a \in \text{Agent}, na \in \text{Nonce}, \\ s \in \text{Server}, nb \in \text{Nonce}, kab \in \text{SessionKey} \}$$

These two constructs, the parameterised deductions and the assignment into one big set, are not correct, as the involved simulated agents must generate the exact specified or mapped variables in the correct combination. For the previously imagined example, the initiator with the identity Alice, that uses Sam as a server will always generate N_p and N_s together within the protocol run, he will never generate $Na1i$ with N_s or N_p with $Na2i$ or try to talk to Bob or Ivo as the Server. On the other side, the initiator will have to accept initially any value for variables of the protocol, that are not in his own argument list.

For the Yahalom protocol in general this means:

- the Initiator needs to be restricted to specific values for a and na and needs to accept initially any b , nb , s , and kab
- the Responder needs to be restricted to specific values for b , s and nb and needs to accept initially any a , na , and kab
- the Server needs to be restricted to specific values for s and kab and needs to accept initially any a , b , na , and nb

Therefore, the deductions are grouped into the several roles of the protocol, e.g. the Yahalom protocol has separate parameterised deductions for the Initiator, Responder and Server.

This will allow to restrict the generations of the individual roles to the exact values, while being able to accept values for the variables

The Parameterised Deductions

The parameterised deductions for every role in the protocol are calculated from the protocol description. The deductions for one role are a set of pairs of a premises set and a conclusion, as outlined in the Section 2.4. For every role deduction set, only those deductions are included whose conclusion represents a sent message of this role.

The exact composition of the deduction, explained in Section 2.4 and originally taken from [4], needs to be carefully outlined to have an exact working model.

For every message of the protocol, one deduction is built. The conclusion of every deduction contains the representation of the message using the sent-constructor. The Tag-part of the sent-construction contains every variable, that the sender of this message has known so far. This includes every variable received by this agent, every variable sent by this agent as well as the identity of the sender and receiver of the message itself.

The set of premises of every deduction contains facts from two different sources:

- all messages that were received by the sender of the current message since he sent the last time a message himself
- the sent-representation of the previous message, that the sender of the current message has sent (if it is not the first message of the agent)

The very first message within the CSP representation of the model, usually $env?b$, is not modelled as its own deduction. Instead it is used for "initiating" the deductions, that means it is the first message received before the very first message of the actual protocol is sent. Consequentially, the set of premises is never empty for any deduction, as every agent will have received one message or sent one, before he sends the next. This is also guaranteed by the existing consistency checks of Casper.

This definition of the necessary premises is a extension of the premises described in [4].

The extension makes deductions for two important variants of protocol messages possible:

- one of the agent receives several messages, before he sends one himself, thus all these received messages need to be considered in the deduction of the next message sent by this agent
- one of the agents sends several messages without receiving one in between, thus the set of premises for these deduction (except the first one) contains only the sent-representation of the previously sent message

The calculation of these deductions was added to Casper in one single function. It calculates every single deduction one after another by going through the protocol description. While doing this, the function keeps track of:

- the received messages of every individual agent
- the last message sent by every individual agent
- the list of variables, that the every individual agent knows

The following are the parameterised deductions calculated for the Yahalom protocol:

$$\begin{aligned}
 & \text{DeductionsINITIATOR}_0(a, na, b, s, nb, kab) \hat{=} \\
 & \left\{ \begin{array}{l} \{b\} \vdash \text{Sent}(na, \langle a, b, na \rangle), \\ \{ \{b, kab, na, nb\}_{\text{ServerKey}(a)}, \text{Sent}(na, \langle a, b, na \rangle) \} \vdash \text{Sent}(\{nb\}_{kab}, \\ \langle a, b, na, kab, nb \rangle) \end{array} \right\} \\
 & \text{DeductionsRESPONDER}_0(a, na, b, s, nb, kab) \hat{=} \\
 & \left\{ \{na\} \vdash \text{Sent}(\{a, na, nb\}_{\text{ServerKey}(b)}, \langle b, s, na, a, nb \rangle) \right\} \\
 & \text{DeductionsSERVER}_0(a, na, b, s, nb, kab) \hat{=} \\
 & \left\{ \begin{array}{l} \{ \{a, na, nb\}_{\text{ServerKey}(b)} \} \vdash \text{Sent}(\{b, kab, na, nb\}_{\text{ServerKey}(a)}, \langle s, a, \\ na, nb, b, kab \rangle), \\ \{ \text{Sent}(\{b, kab, na, nb\}_{\text{ServerKey}(a)}, \langle s, a, na, nb, b, kab \rangle) \} \vdash \\ \text{Sent}(\{ \langle a, kab \rangle \}_{\text{ServerKey}(b)}, \langle s, b, a, na, nb, kab \rangle) \end{array} \right\}
 \end{aligned}$$

The second deduction of the deduction set of the Server simulates a situation, where no previously received messages are part of the premises, because the server sends two messages in a row (Messages 3a and 3b of the Yahalom protocol).

Putting all Deductions Together

The final deductions are composed by instantiating the above depicted deduction-function with the correct assignments.

Basically, two instantiated deduction sets are build:

- One, that instantiates the deduction functions with a restriction to the user-supplied values, therefore doing no mapping at all
- One, that instantiates the deduction functions to mapped values

Deductions with User-supplied Assignments

The first deduction set just instantiates the variables of every role in the protocol with every possible value, but restricts the variables in the argument list of the role to the values provided by the user. Furthermore, these deductions are restricted to honest agents.

The argument lists of the processes in the system are hold as sequences in sets. For example, the set I_{int} holds all arguments of the internal Initiator. For the Yahalom protocol and the example Casper input in Section 3.1.4, the argument sets of the processes are:

$$\begin{aligned} I_{int} &\hat{=} \{ \langle \text{Alice}, \text{Na1i} \rangle, \langle \text{Alice}, \text{Np} \rangle \} \\ R_{int} &\hat{=} \{ \langle \text{Alice}, \text{Sam}, \text{Ns} \rangle \} \\ S_{int} &\hat{=} \{ \langle \text{Sam}, \text{Kabs} \rangle \} \end{aligned}$$

For every individual process of the protocol, the set of the correct assigned deductions is built. As these deduction sets are also used for the compilation optimisation with slightly different sets for the assignment of variables (Section 4.1), these deduction sets are parameterised:

$$\begin{aligned} &DeductionsINITIATOR_{honest}(I) \hat{=} \\ &\bigcup \left\{ \begin{array}{l} \text{deductionsINITIATOR}_0(a, na, b, s, nb, kab) \mid \\ b \in Agent, s \in Server, nb \in Nonce, kab \in \\ SessionKey, \langle a, na \rangle \in I, \text{honest}(a), \text{honest}(b), \text{honest}(s) \end{array} \right\} \end{aligned}$$

$$DeductionsRESPONDER_{honest}(R) \hat{=} \bigcup \left\{ \begin{array}{l} \text{deductionsRESPONDER}_0(a, na, b, s, nb, kab) \mid \\ a \in Agent, na \in Nonce, kab \in SessionKey, \langle b, s, nb \rangle \in \\ R, \text{honest}(a), \text{honest}(b), \text{honest}(s) \end{array} \right\}$$

$$DeductionsSERVER_{honest}(S) \hat{=} \bigcup \left\{ \begin{array}{l} \text{deductionsSERVER}_0(a, na, b, s, nb, kab) \mid \\ a \in Agent, na \in Nonce, b \in Agent, nb \in Nonce, \\ \langle s, kab \rangle \in S, \text{honest}(a), \text{honest}(b), \text{honest}(s) \end{array} \right\}$$

The complete set of all deductions of all processes, initiated with the correct argument sets, is then:

$$\text{InternalDeductions}_{honest} = \bigcup \left\{ \begin{array}{l} \text{DeductionsINITIATOR}_{honest}(I_{int}), \\ \text{DeductionsRESPONDER}_{honest}(R_{int}), \\ \text{DeductionsSERVER}_{honest}(S_{int}) \end{array} \right\}$$

Deductions with Mapped Assignments

The second set, the mapped deductions are more complicated, they depend on the activation or deactivation of the variable *UnboundOptimize*.

When the option *UnboundOptimize* is set to false, the mapped deductions are restricted to sessions, that involve the intruder. The parameterised deductions are basically the same as the previous, with the difference, that all generated variables are set to T_p :

$$DeductionsINITIATOR_{mapped}(I) \hat{=} \bigcup \left\{ \begin{array}{l} \text{deductionsINITIATOR}_0(a, \mathbf{Np}, b, s, nb, kab) \mid \\ b \in Agent, s \in Server, nb \in Nonce, kab \in \\ SessionKey, \langle a, _ \rangle \in I, \text{not} (\text{honest}(a) \wedge \text{honest}(b) \wedge \\ \text{honest}(s)) \end{array} \right\}$$

If the option *UnboundOptimize* is used, then no fixed value is used for generated variables, instead a mapping function (the secrecy decision) is used:

$$DeductionsINITIATOR_{mapped}(I) \hat{=} \bigcup \left\{ \begin{array}{l} \text{deductionsINITIATOR}_0(a, \text{decide_secrecy_na}(a, b, s), \\ b, s, nb, kab) \mid \\ b \in Agent, s \in Server, nb \in Nonce, kab \in \\ SessionKey, \langle a, _ \rangle \in I \end{array} \right\}$$

This set does not contain the constraint to sessions with at least one dishonest agent. The `decide_secrecy` function is explained in the next section. It will return T_p , if one of the agents of the session is the intruder and T_p or T_s based on the secrecy of the variable, if all agents are honest.

All deductions of the internalised agents are then put together into one set as:

$$InternalDeductions \hat{=} \bigcup \left\{ \begin{array}{l} InternalDeductions_{honest}, \\ DeductionsINITIATOR_{mapped}(I_{int}), \\ DeductionsRESPONDER_{mapped}(R_{int}), \\ DeductionsSERVER_{mapped}(S_{int}) \end{array} \right\}$$

All deductions, to be used with the `Close`-function are then:

$$BaseDeduction \hat{=} CoreDeductions \cup InternalDeductions \cup SentDeduction$$

3.3 Secrecy Decision

In Section 2.4.2, a collapsing function was outlined, that applies different mappings to variables, that are secret and those that are public. This mapping is based on the apparent secrecy of the variables, i.e. of someone not involved in the protocol run might be able to get the value that is used for the variable.

An example is here message 1 of the Yahalom protocol:

Message 1. $a \rightarrow b : na$

The variable na is not encrypted, therefore clearly public.

It might be possible, that a variable is public in an indirect way. For example, the following messages of an hypothetical protocol contain two public variables:

Message 1. $a \rightarrow b : kab$

Message 2. $b \rightarrow a : \{na\}_{kab}$

The variable kab is sent unencrypted and therefore public, the variable na is then sent encrypted with the key kab . As it is easily possible to decrypt na with the previously sent

key kab , na variable is also public.

Therefore, the best possible secrecy decision will need to consider all conclusions that can be drawn from the available knowledge.

This can be done by using the deduction system, which can automatically decrypt messages with the available knowledge.

The nature of the check requires, that for the secrecy decision the concerning protocol is run only with honest agents, i.e. only those deductions are used that involve honest agents or the intruder pretending to be the honest agent. It is important, that the identity of the intruder is not used in the protocol runs, as one of the agents might encrypt values with keys, that are clearly known to the intruder. This would of course not reveal the general secrecy of the concerning variable correctly.

The secrecy of variables is checked by using the deductions of all internal honest agents together with the core deductions outlined in Section 2.3.3 and the `Close()` function. The used deductions are composed as:

$$\text{AllInternalHonestDeductions} = \text{CoreDeductions} \cup \text{InternalDeductions}_{\text{honest}}$$

These deductions are then used to calculate the knowledge that is accessible from this:

$$\text{IK}_{\text{Sec}} = \text{Close}_{\text{Fact}_1}^{\text{AllInternalHonestDeductions}}(\text{IK}_0)$$

For every variable that is generated and therefore needs a mapping based on a secrecy decision, one function is generated in the CSP code, that takes the participating agents as argument and returns the variable that is to be used. If one of the participating agents is not honest, the standard value with the status *InternalKnown* is returned.

If a representative value (i.e. one value that is assigned to this variable in one of the internal agents) is in the set IK_{Sec} , the variable is public and the standard value with the status *InternalKnown* is returned, too. In any other case, the standard variable with the status *InternalUnknown* is used.

The following function was taken from the Yahalom protocol, it calculates the mapped values for the variable na :

```
decide_secretcy_na(a, b, s) =
    if not (honest(a) and honest(b) and honest(s))
    then Np
    else (if member(Na1i, IKSec) then Np else Ns)
```

The function therefore does not only calculate the mapping for the interaction of honest agents, but also the mapping for any communication with the intruder.

3.4 Type Checks and Consistency

The current implementation of Casper already contains a wide range of type and consistency checks. These shall for example assure, that the user-supplied Casper input does not contain contradictions or breaks of general rules and that the user has supplied all necessary values.

Some checks had to be added for the implementation of the unbounded parallel mode. These can be found in the added Haskell file *TypeCheckUP.lhs*.

3.4.1 Additional Checks for the Original Model

As it was necessary to introduce new keywords and systematics for the unbounded parallel mode, the absence of the usage of these keywords is checked, if the unbounded parallel mode is disabled.

The following checks are currently done in this situation:

- The *UnboundOptimize* mode must not be enabled. Although the *UnboundOptimize* mode can be used to speed up compilation of the bound mode, the unbounded parallel mode needs to be enabled explicitly, to enable the use of the internalisation of agents.
- The *DecideSecrecy* option must not be set to true, as its use is only necessary in the unbounded parallel mode
- No variable in the section *Actual variables* is allowed to have one of the statuses associated with the unbounded parallel mode (see Section 3.1.3): *Internal*, *External*, *InternalUnknown*, or *InternalKnown*.
- No actual agent in the *System* section is allowed to be declared as *internal* (see Section 3.1.4)

The *generate*-keyword, used in the *Processes* section is also used for the Data Independence mode, so that its presence is not a sign of a mistake in the protocol specification.

3.4.2 New Checks for the Unbounded Parallel Model

The unbounded parallel model puts some restrictions on the user and requires some further adjustments. Furthermore, the implementation of the unbounded parallel model does not support every feature of the original Casper implementation for internalised agents, so that the usage of some of these features throws an error, too.

The following checks are done:

- For every type that is used for generated variables, one value with the status *InternalKnown* must be declared. This is necessary, because this will be the value that is generated when running the protocol with the intruder.

- If the *DecideSecrecy* option is enabled, there must be a value with the status *InternalUnknown* for every type, that is used for generated variables. This is necessary, because the function that decides the secrecy must give this value back, if the variable is apparently secret.
- The protocol description must not contain any percentage notation.
- Every variable that is declared to be generated must also appear in the argument list of the process, that generates it.

3.4.3 Unimplemented Checks

There several additional consistency checks necessary, that have not yet been implemented. All but the last two of these consistency checks for the unbounded parallel mode apply to the instantiation of generated variables in the actual agents.

- Every generated variable must be instantiated by an value, that has one of the statuses *Internal*, *External*, *InternalUnknown*, or *InternalKnown*.
- External agents must only use values with the status *External* for generated variables.
- Internal agents must either only be instantiated with values with the status *Internal* or only values with the statuses *InternalUnknown* or *InternalKnown* for every generated variable.
- If the option *DecideSecrecy* is enabled, generated variables must not be instantiated with values, that have the status *InternalUnknown* or *InternalKnown*.
- If the option *DecideSecrecy* is disabled, at least on internal agent for every process must be instantiated with only values with either the status *InternalUnknown* or the status *InternalKnown* for generated variables.
- The intruder's initial knowledge must contain a value of the status *InternalKnown* for every type, that is used for generated variables.
- There is only one value declared with the status *InternalKnown* and a maximum of one with the status *InternalUnknown* for every type, that is used for generated variables.

3.4.4 Unimplemented Warnings

In Section 3.1 it was established how Casper input needs to be configured. Future additions to the implementation might be able to configure agents and values without the user input automatically. Until then, the compilation of a script should result in a warning, if it contains specifications that might not result in the desired outcome or will be slower then necessary.

There are many possible warnings. The following are some of the most useful ones, until the future extension is available:

- The option `DecideSecrecy` should not be enabled, if a secrecy analysis is done, instead, the user should only specify internal agents that instantiate generated variables with values that have the status *InternalKnown* or *InternalUnKnown*.
- Similarly, no external agent should be specified if a secrecy analysis is done.

4 Optimisations

The internalisation of agents and the additional values in the system increase the complexity of the calculation of the model. Eldar Kleiner found in [3] several ways to increase the performance of the implementation of the unbounded parallel model in CSP.

FDR checks a model in a two step system: It first compiles the machine-readable CSP script into an internal representation and then runs the state space check on it. The state space was already reduced considerably through the unbounded model alone, as the internal deductions can be computed very fast. For example, the secrecy analysis is reduced to a static check.

But more major improvements are possible. Most of them concern the number of compilation states but one was done for the the actual state space check.

The compilation can be improved by accomplishing the following means:

- Decreasing the number of renamings
- Decreasing the complexity of computations
- Decreasing the number of processes in the system
- Decreasing the size of the state space of the single processes (for example by decreasing the possible choices)

4.1 Decreasing the Number of Learnable Facts

Knowable Facts

In Section 2.3.3, the calculation of the sets IIK_1 , $Deductions$, and $KnowableFact$ was explained and how the number of fact processes within the intruder is determined.

The set $KnowableFact$ is an over-approximation, consisting of conclusions of the remaining inferable deductions and the observable messages. This is far too much, as it contains messages that never appear (e.g. messages with impossible variable-generations).

One example is the fact $\{Ivo, Kabs\}_{ServerKey(Alice)}$ in the Yahalom protocol, that cannot occur because the Server Sam would not (by definition) willingly issue the secret key $Kabs$ in a session that involves the intruder Ivo . Furthermore, the intruder cannot deduce this message, because $ServerKey(Alice)$ is never sent in any message, so that Ivo can never encrypt this example message himself. Therefore, the intruder contains a large number of processes, that can never reach the knowing state and fulfill no purpose.

It was therefore necessary to find a better way to approximate $KnowableFact$. This set must at least contain every fact that can be known. It is therefore possible to over-approximate this set, but it must not be under-approximated. The clear aim is therefore, to have an approximation, that is as close as possible, but that contains provable every knowable fact.

Using the Internalisation of External Agents as an Over-approximation

The internal deductions can help here. The deductions *InternalDeductions* contain the deduction system to simulate the behaviour of the internal agents.

It is possible to internalise the external agents in the same way just for the purpose of calculating *KnowableFact*. This simulation is not sufficient to be used for the security flaw checks, but they suffice as an over-approximation of the behaviour of the external behaviour.

The deductions for all agents, internal and external are then defined for the Yahalom protocol as:

$$\text{InternalExternalDeductions} = \bigcup \left\{ \begin{array}{l} \text{InternalDeductions}, \\ \text{DeductionsINITIATOR}_{\text{honest}}(I_{\text{ext}}), \\ \text{DeductionsRESPONDER}_{\text{honest}}(R_{\text{ext}}), \\ \text{DeductionsSERVER}_{\text{honest}}(S_{\text{ext}}) \end{array} \right\}$$

I_{ext} , R_{ext} , and S_{ext} are the configuration sets of the external Intruders, Responders and Servers, i.e. the list of their arguments including the generated variables. Furthermore, the unmapped deduction functions are used, so that the internalised external agents will not run the protocol with an agent, that uses the identity of the intruder. This is due to another restriction on the behaviour of the processes of the external agents to reduce the state space (explained in Section 4.3).

The set *KnowableFact* can then be computed as the knowledge that can be deduced from the intruder's initial knowledge IIK_0 , using the above deductions and the in section 2.3.3 defined deductions. This will simulate an infinite parallel run of all instances, internally and externally including all messages that might every occur.

For this to work, the Close function had to be altered. The calculation of the third result of Close (*KnowableFact*) uses knowledge about the message structure and content. The sets that contain the structure and content of the messages was optimised (see Section 4.2) to be based on knowledge from *KnowableFact*, so that a loop in the calculation would occur. As the third result in this calculation is not needed, it was dropped entirely from the function.

The new set *KnowableFact* is then calculated as:

$$(\text{KnowableFact}, _) = \text{Close}_{\text{Fact}_1}^{\text{InternalExternalDeductions} \cup \text{CoreDeductions}}(IIK_0)$$

The then derived set *LearnableFact* is remarkably smaller, as it could be shown in the benchmarks in Chapter 5.

KnowableFact can then also be used to decrease the complexity of the computation of IIK_1 . IIK_1 must be a subset of *KnowableFact*, so that IIK_1 can be calculated on the basis of the

set *KnowableFact* instead of *Fact₁*:

$$(IIK_1, \text{Deductions}) = \text{Close}_{\text{KnowableFact}}^{\text{BaseDeductions}}(IIK_0)$$

As *KnowableFact* is considerably smaller than its superset *Fact₁*, the above depicted computation is faster than the original one.

Of course, now it is necessary to compute the Close function twice instead of only once, but this drawback is easily overcome by the performance improvement in the compilation of the fact-processes.

This optimisation technique will be automatically activated by enabling the *Unbound-Optimize* option in the Casper input file. The implementation required generating the above mentioned deductions that simulate the internal and external agents as well as the substitution of the original single Close-statement by the two Close-executions to build *KnowableFact* and *IIK₁* separately.

Removing the sent-constructor of the facts of external agents

The simulation of external agents to get an over-approximation of *KnowableFact* still yields more facts than necessary. For example, it will include facts using the sent-constructor, that only appear in protocol runs with the external agents. Of course, these sent-facts will then never be learned, as the sent-constructor is only used for internal agents, when the intruder tries to actually get new knowledge.

The sent-constructor was introduced to restrict the deductions that simulate the internal agents. The sent-fact shall prohibit the internalised agents from showing behaviour, that they would not have externally. This restriction is not necessary as *KnowableFact* shall be over-approximated.

Therefore, the deductions, that are used to simulate the external agents do not need to use the sent-constructor. It might be possible, that the resulting *KnowableFact* will include some unnecessary facts, even facts that are wrong in the sense that it is not possible to get them in a protocol run with the external agent, but the decrease in the number of sent-facts should outperform the drawback.

This feature was implemented by adding a copy of every deduction of the internal agents to the system. But instead of using the construct $\text{Sent}(M, T)$, only M was used in the conclusion and in the premises of the deduction.

For example, the deduction for the server in the Yahalom protocol becomes:

$$\text{Deductions}_{\text{SERVER}_{\text{external0}}}(a, na, b, s, nb, kab) \hat{=} \left\{ \begin{array}{l} \{ \{a, na, nb\}_{\text{ServerKey}(b)} \} \vdash \{b, kab, na, nb\}_{\text{ServerKey}(a)}, \\ \{ \{b, kab, na, nb\}_{\text{ServerKey}(a)} \} \vdash \{ \langle a, kab \rangle \}_{\text{ServerKey}(b)} \end{array} \right\}$$

By using this method, some more learnable facts could be removed, thus the performance increased. For example, for the check of injective authentication of the Yahalom protocol, the cardinality of *LearnableFact* could be decreased from 26 to 22.

4.2 Decreasing the Size of the External Agents

FDR will generate an internal representation of every process that is used in the current specification check. The number of compilation states to calculate this internal representation also depends on the possible states, that the process can take. This is directly influenced by the choices done by the process.

As an example, this is again the initiator of the Yahalom protocol:

$$\begin{aligned} \text{Initiator}(a, na) \hat{=} & \text{env?}b : \text{Agent} \rightarrow \\ & \text{send.a.b.na} \rightarrow \\ & \square_{kab \in \text{Key}, nb \in \text{Nonce}, s \in \text{Server}} \left(\begin{array}{l} \text{receive.s.a.}\{a, kab, na, nb\}_{\{\text{ServerKey}(a)\}} \rightarrow \\ \text{send.a.b.}\{nb\}_{\{kab\}} \rightarrow \text{Session}(a, b, na, nb, kab) \end{array} \right) \end{aligned}$$

Although the choices that this process can make are entirely correct, this process alone will accept events that the process in parallel to the network will never be able to do. For the compilation, FDR calculates all possible states of this process and does not limit it to the possible states under the restrictions of outside processes.

The process $\text{Initiator}(\text{Alice}, Na)$ of the Yahalom protocol will never have to accept the event $\text{receive.Sam.Alice.}\{\text{Alice}, Kabs, Na, Np\}_{\{\text{ServerKey}(\text{Alice})\}}$ (corresponding to receiving message 3a), as this message can only be generated by the Server, if he thinks that all involved agents are honest. But in this case the variable nb would be initiated with Nb (externally generated) or Ns (internally generated) by the Responder. The intruder himself is not able to fake this message 3a or the message 2 that would cause this message 3a. Yet, the above depicted process does not make this restriction.

The CSP scripts generated by Casper already define a set for every message, that contains every possible variable-assignment and some extra information, that are used for correct renamings. The set for message 3a in the original implementation is called INT_MSG_INFO3a . For the implementation of this compilation optimisation, the name was changed to $INT_MSG_INFO3a_0$, so that the new and smaller set can have the original name:

$$\begin{aligned} INT_MSG_INFO3a_0 \hat{=} & \\ & \bigcup \left\{ \begin{array}{l} (\text{Msg3a}, \text{receive.s.a.}\{a, kab, na, nb\}_{\{\text{ServerKey}(a)\}}, \langle \rangle, \langle \rangle) \\ | \\ a \in \text{Agent}, b \in \text{Agent}, kab \in \text{SessionKey}, na \in \\ \text{Nonce}, nb \in \text{Nonce} \end{array} \right\} \end{aligned}$$

This set can be restricted to the messages, that are really possible to occur. As shown in the previous section, the set *KnowableFact* is a valid (yet over-approximated) restriction to the possible messages.

The set of messages, that can really occur is then easily defined as:

$$INT_MSG_INFO3a \hat{=} \bigcup \left\{ \begin{array}{l} (Msg3a, m, s, r) \mid \\ (Msg3a, m, s, r) \in INT_MSG_INFO3a_0, m \in \\ KnowableFact \end{array} \right\}$$

The variables *s* and *r* are extra sender and receiver information, that are needed in the specifications, they are of no concern at this point.

It might be desirable to restrict these messages even to messages from the set *LearnableFact*, as the intruder is interested especially in them. But, even a protocol run that yields messages from *LearnableFact* will include some messages from the superset *KnowableFact*. Especially the content of the first message, the agent with which to start the protocol, will be known to the intruder and therefore not be in *LearnableFact*. Therefore, the messages need to be restricted at the moment to *KnowableFact*. But it might be desired to change future implementations to restrict the external agents to protocol runs, that involve at least one learnable fact.

It must be noted, that the restriction of the agents to the set *KnowableFact* does not change the behaviour of the process in the context of the restrictions that the whole system puts on the agent. But a restriction to protocol runs that involve at least one message from *LearnableFact* will decrease the number of states and transitions in the actual check.

But not the general usable message set *INT_MSG_INFO3a* is used, instead the separate sets *OUTPUT_INT_MSG3a* for send and *INPUT_INT_MSG3a* for received messages are used. This is necessary, as only the extra send information (the third part of the message tuple in the example) are available for the send event and the extra receiver information for the receive event. The sets *OUTPUT_INT_MSGn* and *INPUT_INT_MSGn* are appropriate mappings of *INT_MSG_INFOn* to these information.

The definition of the processes then also has to be changed, so that only messages are accepted, that are from these sets. Fortunately, this feature already existed for the implementation of the Data Independence model.

The old machine-readable definition of message 3a as seen by the initiator of the Yahalom protocol was:

```
[] kab : SessionKey @ [] nb : Nonce @ [] s : Server, INTRUDER_M::honest(s) @
receive.s.a.(Msg3a, Encrypt.(ServerKey(a), <b, kab, na, nb>), <>) ->
```

It is changed to:

```
[] kab : SessionKey @ [] nb : Nonce @ [] s : Server, INTRUDER_M::honest(s) @
true and
member((Msg3a, Encrypt.(ServerKey(a), <b, kab, na, nb>),<>), INPUT_INT_MSG3a) &
receive.s.a.(Msg3a, Encrypt.(ServerKey(a), <b, kab, na, nb>),<>) ->
```

Note that the actual usage of the receive and send event in the machine readable CSP is slightly different from the notation of this dissertation, as the message events contains some more information than just the message itself.

The CSP construct

B & P

describes a process with the meaning "if B then P else STOP". If the condition B is satisfied, then the overall process behaves like P, else like STOP. The usage of the construct in the agent-definitions restricts the agent to only knowable facts.

Both incoming and outgoing messages have to be restricted, as some outgoing messages could, although restricted by the values of previous choices on the events and without the restriction of other parallel processes be outside of *KnowableFact*. This would lead to errors in FDR during the renaming of some of the events of the external agents.

This compilation optimisation feature is activated by enabling the option *UnboundOptimize*.

4.3 Excluding the Identity of the Intruder from External Protocol Runs

As stated in [3], it is not necessary to run the protocol with external agents with the identity of the intruder.

This has the effect of reducing the state space of the model as well as a compilation optimisation, as the intermediate representation that needs to be compiled is smaller.

This limitation is implemented in the definitions of the external processes as well as in the internalisation of the external processes for the purpose of calculating the set *KnowableFact*.

For example, the definition of the initiator in the Yahalom protocol contains the identities of agents *a* and *b*, that need to be prevented from taking the values of the intruder (*Ivo*, in this example). The CSP definition of the initiator of the Yahalom protocol gets then changed to:

$$\begin{aligned}
\text{Initiator}(a, na) = & \square_{\mathbf{b} \in \text{Agent} \setminus \{Ivo\}} \text{env?}\mathbf{b} \rightarrow \\
& \text{send.a.}\mathbf{b}.na \rightarrow \\
& \square_{kab \in \text{Key}, nb \in \text{Nonce}, s \in \text{Server} \setminus \{Ivo\}} \text{receive.s.a.}\{a, kab, na, nb\}_{\{\text{ServerKey}(a)\}} \rightarrow \\
& \text{send.a.}\mathbf{b}.\{nb\}_{\{kab\}} \rightarrow \text{Session}(a, \mathbf{b}, na, nb, kab)
\end{aligned}$$

In the machine-readable CSP, this gets implemented by adding a restriction to the generalised choice. In this case, a function was defined which returns true if the agent is honest and false, if it is not honest. The following example shows the first event:

```
[| b : Agent, INTRUDER_M::honest(b) @ env_I.a.(Env0, b, <>) -> ...
```

This change decreased the number of checked states from 196 to 169 and transitions from 616 to 533 for the check of the injective authentication of the Yahalom protocol (see Section 5.2.1).

This optimisation was implemented independently of the activation of the option *UnboundOptimize* in the Casper input, since this option should only influence the compilation of the CSP script and not the size of the state-space.

4.4 Decreasing the Number of Renamings

Some time of the compilation of CSP scripts in FDR is taken by the renamings of events. Decreasing the number of necessary renamings brings not only a performance increase, it is also necessary for completing the implementing the decrease of the size of the external agents in Sections 4.2 and 4.3.

FDR will throw an error, if events are renamed, that the process cannot communicate¹.

This renaming problem appears for the authentication specification checks and not in the secrecy checks, as the latter one was changed completely.

In the section for authentication checks, Casper creates a process system, whose events that concern the authentication are renamed appropriately (see 2.4.5 for details on how authentication checks are done). Furthermore, all events not involved in the authentication check are hidden.

The following renamings and hidings CSP are used for the initiator to responder authentication check in the Yahalom protocol (some details were removed, `alphaAuthenticateINITIATORToRESPONDERAgreement_na_nb` is the alphabet of the specification process):

¹Specific error: *readEventMap failed*

$$\begin{aligned}
& [[\text{send.a.b.}(\text{Msg4}, \{\text{nb}\}_{\text{kab}}, \langle \text{na}, \text{nb} \rangle) \leftarrow \text{Running.a.b.na.nb} \\
& \quad \text{receive.a.b.}(\text{Msg4}, \{\text{nb}\}_{\text{kab}}, \langle \text{na}, \text{nb} \rangle) \leftarrow \text{Commit.a.b.na.nb} \\
& \quad | a \in \text{Agent}, b \in \text{Agent}, na \in \text{Nonce}, nb \in \text{Nonce}, kab \in \text{SessionKey} \\
& \quad]] \setminus (\Sigma \setminus \alpha \text{AuthenticateINITIATORToRESPONDERAgreement_na_nb})
\end{aligned}$$

Here, every possible assignment of the variables of the send and receive command are renamed to the proper running and commit signal events. This is far too much, as the possible receive and send events were restricted using several methods, as explained in the previous sections.

The solution is easy, the concerning messages must be member of the message sets, that were used previously for the process restrictions. The restriction of the renamings to actually possible events is:

$$\begin{aligned}
& [[\text{send.a.b.}(\text{Msg4}, \{\text{nb}\}_{\text{kab}}, \langle \text{na}, \text{nb} \rangle) \leftarrow \text{Running.a.b.na.nb} \\
& \quad \text{receive.a.b.}(\text{Msg4}, \{\text{nb}\}_{\text{kab}}, \langle \text{na}, \text{nb} \rangle) \leftarrow \text{Commit.a.b.na.nb} \\
& \quad | a \in \text{Agent}, b \in \text{Agent}, na \in \text{Nonce}, nb \in \text{Nonce}, kab \in \text{SessionKey}, \\
& \quad \text{member}((\text{Msg4}, \{\text{nb}\}_{\text{kab}}, \langle \text{na}, \text{nb} \rangle), \text{OUTPUT_INT_MSG4}), \\
& \quad \text{member}((\text{Msg4}, \{\text{nb}\}_{\text{kab}}, \langle \text{na}, \text{nb} \rangle), \text{INPUT_INT_MSG4}) \\
& \quad]] \setminus (\Sigma \setminus \alpha \text{AuthenticateINITIATORToRESPONDERAgreement_na_nb})
\end{aligned}$$

The two restrictions to the messages above seem at first redundant, but they are necessary if not the same messages are renamed. Furthermore, both the sender and receiver extra information need to be correct for the renaming to be allowed.

Other authentication checks might involve other messages, the restrictions are modelled accordingly.

It is difficult to give a prediction of the performance outcome of this change, as the implementation of this feature is necessary for the optimisation of the external agents to work.

This feature is automatically enabled by the option *UnboundOptimize*.

4.5 Minimising Fact₁

The size of Fact₁ has a direct influence on the calculation time of $\text{Close}_{\text{Fact}_1}^{\text{Ded}}$. It was shown before, how the set KnowableFact can be used as a substitute for Fact₁ to reduce the calculation time.

But Fact_1 can also be reduced by limiting the number of sent-facts in it. Initially, Fact_1 contains every possible assignment of variables for the sent-facts. Some of them will never appear as fact in any of the deductions, as the variables sent-facts of messages that generate values are restricted to specific values.

For example, since the sent-constructor is not used for the deductions of external agents, a nonce of an external agent cannot be generated by a sent-deduction and does not need to be included in Fact_1 .

Limiting the sent-facts in Fact_1 to the absolute necessary is quite easy, as all possible sent-facts are defined by the conclusions of the internal deductions. Therefore, the set of sent-facts that have to be included in Fact_1 will be defined as:

$$\text{Sentfacts} = \{ \text{sm} \mid \exists x \bullet (\text{sm}, x) \in \text{InternalExternalDeductions} \}$$

The reduction in compilation time of the scripts are very small, but it may help in the future with even bigger fact-sizes. This feature is activated, when the user requests the compilation optimisation but not the secrecy decision. The latter option must be disabled, because for the secrecy decision the original Fact_1 is used to compute the correct sent-deductions.

5 Benchmark results

The following tables contain the benchmark results of several different protocols checked by the implementation of the unbounded model of this dissertation in comparison with the performance of the CSP models produced by Casper 1.8.

For the unbound model, secrecy, non-injective authentication and injective authentication were checked. One internal initiator, responder and if applicable one internal server were used for the secrecy analysis. For non-injective authentication, two internal initiators, one internal and one external responder and one internal server, if applicable, were used. For the injective authentication, one extra external responder was added.

These numbers for internal and external initiators, responders and servers were taken from Section 3.1.6. The unbounded model is complete, so that if no attack is found with the defined configuration, then no attack upon the protocol (under the Dolev-Yao assumption) exists.

The identity "Alice" was used for all responders and initiators.

All checks were done with compilation optimisation and without. In Section 2.7.3 of [3] an additional reduction of the intruder's knowledge size was proposed by removing some unnecessary sent-facts. As the sent-constructor is used to restrict the behaviour of internal agents, it is not necessary to use it for the deduction that models the last message of the agent. The sent-constructor can therefore be removed easily for these deductions, if these messages are not used in the authentication renaming. For some protocols, this removal was tested manually.

The check with the traditional bounded "Alice and Bob"-model was done with Alice as initiator, Bob as responder and Sam as server, if necessary. The bounded model is not complete, so that there might be still security flaws in the protocol, even if none was found.

5.1 Legend

The following values were determined:

- Was an attack found?
- The number of generated values in the systems (nonces + keys)
- The number of explored states
- The number of explored transitions
- The number of compilation transitions
- The number of compilation transitions with compilation optimisation for the unbound model
- The number of the learnable facts (`|LearnableFact|`)

- The number of learnable facts, with compilation optimisation
- The number of learnable facts with a manually reduced number of Sent-facts
- The number of learnable facts with compilation optimisation and a reduced number of Sent-Facts

One requirement for the *UnboundOptimize* option was, that the number of explored states and transitions does not depend on its activation or deactivation. This was checked for every test run and the requirement was fulfilled.

A dash was used to denote an entry that is not applicable and a question mark for values that were not determined.

5.2 Results

5.2.1 Yahalom Protocol

There is no known attack upon the Yahalom protocol.

	Unbounded parallel Model			Original Model
	Secrecy Check	Non-Injective Auth	Injective Auth	
Attack found?	No	No	No	No
Number of values	4	6	7	4
Number of explored states	1	13	169	1731
Number of explored transitions	1	27	533	7561
Number of compilation transitions (w/o compilation optimisation)	600	2600	4300	1900
Number of compilation transitions (w/ compilation optimisation)	1	100	200	-
LearnableFact	80	306	497	174
LearnableFact (w/ reduced Sent-Facts)	66	250	403	-
LearnableFact (w/ comp. optimisation)	0	11	22	-
LearnableFact (w/ comp. optimisation and reduced Sent-Facts)	0	9	18	-

The Yahalom protocol shows the most improvement for the compilation speed-up. The number of learnable facts could be decreased from 497 to 18 for the injective authentication with seven values.

It seems that the optimisation technique works the best for correct protocols, as most of the protocol's message space does not need to be explored. This can be seen in the analysis

of the Yahalom and Needham/Schroeder/Lowe Public Key protocol in comparison with the highly flawed TMN protocol.

5.2.2 Needham/Schroeder Public Key Protocol

The Needham/Schroeder Public Key has known secrecy and authentication attacks. They were first discovered in [5].

	Unbounded parallel Model			Original Model
	Secrecy Check	Non-Injective Auth	Injective Auth	Secrecy Check
Attack found?	Yes	Yes	Yes	Yes
Number of values	2	4	5	3
Number of explored states	1	8	41	114
Number of explored transitions	0	16	148	420
Number of compilation transitions (w/o compilation optimisation)	1	400	800	500
Number of compilation transitions (w/ compilation optimisation)	1	200	500	-
LearnableFact	0	46	76	50
LearnableFact (w/ reduced Sent-Facts)	0	43	71	-
LearnableFact (w/ comp. optimisation)	0	23	50	-
LearnableFact (w/ comp. optimisation and reduced Sent-Facts)	0	21	46	-

5.2.3 Needham/Schroeder/Lowe Public Key Protocol

The Needham/Schroeder/Lowe Public Key protocol is a proposed variant of the Needham Schroeder Public Key protocol. There is no known attack upon this protocol.

	Unbounded parallel Model			Original Model
	Secrecy Check	Non-Injective Auth	Injective Auth	Secrecy Check
Attack found?	No	No	No	No
Number of values	2	4	5	3
Number of explored states	1	11	121	112
Number of explored transitions	1	21	341	413
Number of compilation transitions (w/o compilation optimisation)	100	900	1400	1000
Number of compilation transitions (w/ compilation optimisation)	1	101	100	-
LearnableFact	18	88	136	50
LearnableFact (w/ reduced Sent-Facts)	17	84	130	-
LearnableFact (w/ comp. optimisation)	0	6	12	-
LearnableFact (w/ comp. optimisation and reduced Sent-Facts)	0	6	12	-

5.2.4 BAN simplified version of the Yahalom protocol

The BAN simplified version of the Yahalom protocol has known authentication attacks.

	Unbounded parallel Model			Original Model
	Secrecy Check	Non-Injective Auth	Injective Auth	Non-Injective Auth
Attack found?	No	Yes	Yes	No
Number of values	4	6	7	4
Number of explored states	1	17	140	2618
Number of explored transitions	1	59	871	6223
Number of compilation transitions (w/o compilation optimisation)	900	3400	4500	1600
Number of compilation transitions (w/ compilation optimisation)	1	800	2800	-
LearnableFact	92	317	424	57
LearnableFact (w/ reduced Sent-Facts)	78	265	356	-
LearnableFact (w/ comp. optimisation)	0	96	288	-
LearnableFact (w/ comp. optimisation and reduced Sent-Facts)	0	74	220	-

5.2.5 Otway Rees Protocol

The Otway Rees protocol has known authentication attacks.

	Unbounded parallel Model			Original Model
	Secrecy Check	Non-Injective Auth	Injective Auth	Secrecy Check
Attack found?	No	Yes	Yes	*
Number of values	4	6	7	4
Number of explored states	1	6	14	6412
Number of explored transitions	1	11	53	16229
Number of compilation transitions (w/o compilation optimisation)	500	1900	3300	1100
Number of compilation transitions (w/ compilation optimisation)	1	600	1500	-
LearnableFact	56	196	327	52
LearnableFact (w/ reduced Sent-Facts)	?	?	?	-
LearnableFact (w/ comp. optimisation)	0	64	148	-
LearnableFact (w/ comp. optimisation and reduced Sent-Facts)	?	?	?	-

* No secrecy attack was found and the authentication attack was found.

5.2.6 Neumann Stubblebine

The Neumann Stubblebine protocol consists of two sub-protocols. The first part is used to authenticate agent a to agent b via an authentication server and for giving a ticket to a , that is used for reauthentication in the second part of the protocol.

The ticket consists of the identity of a , a key kab and of a timestamp. Timestamps are not supported by the unbounded model currently, so that any appearance of a timestamp was removed from the protocol for all tests.

This protocol is a good case study for the scalability of the unbound model. The non-injective authentication of the first part of the protocol is checked, while the second part runs in the background.

As the nonces na and nb are both public in this protocol, the variable Ns is not used. The test with five values does not contain it in the declaration of actual variables while the test with six values does.

The protocol itself has known authentication attacks, which could be confirmed.

	Unbounded parallel Model	
	Non-Injective Auth 1	Non-Injective Auth 2
Attack found?	Yes	Yes
Number of values	5	6
Number of explored states	5	5
Number of explored transitions	12	12
Number of compilation transitions (w/o compilation optimisation)	1500	3300
Number of compilation transitions (w/ compilation optimisation)	900	900
LearnableFact	160	325
LearnableFact (w/ reduced Sent-Facts)	?	?
LearnableFact (w/ comp. optimisation)	98	98
LearnableFact (w/ comp. optimisation and reduced Sent-Facts)	?	?

This benchmark shows, that the definition of an unnecessary value has a negative impact on the performance of the compilation and that the compilation optimisation compensates this problem.

5.2.7 TMN Protocol

The TMN protocol has known secrecy and authentication attacks.

	Unbounded parallel Model			Original Model
	Secrecy Check	Non-Injective Auth	Injective Auth	Secrecy Check
Attack found?	Yes	Yes	Yes	*
Number of values	2	4	5	3
Number of explored states	1	2	2	1066
Number of explored transitions	0	2	3	3382
Number of compilation transitions (w/o compilation optimisation)	1	500	1100	500
Number of compilation transitions (w/ compilation optimisation)	1	500	1100	-
LearnableFact	0	57	128	13
LearnableFact (w/ reduced Sent-Facts)	?	57	128	-
LearnableFact (w/ comp. optimisation)	0	57	128	-
LearnableFact (w/ comp. optimisation and reduced Sent-Facts)	?	57	128	-

* The authentication attack was found by the bounded model with one intruder, one responder and one server. The secrecy attack was not found in the bounded model. The secrecy attack was found by internalising the intruder using the Data Independence model (Broadfoot [1]). This was possible as the server does not introduce fresh values.

The TMN protocol is highly flawed, which explains the missing improvement in the number of learnable facts between the tests with and without compilation optimisation. Because of the protocol’s flaws, it is not possible to rule out that certain facts might be learned by the intruder, so that these fact processes must be included in the intruder.

5.3 Discussion

The unbound model provides a proof of the correctness of protocols with no restriction on the number of parallel sessions and for an unbounded message space. All attacks found upon protocols were correct. The benchmarks show, that the new model can also decrease the number of explored states and transitions considerably. Furthermore, using several compilation optimisation techniques, the compilation time can be reduced dramatically.

One major factor that increases the number of compilation transitions is the number of values in the system. The compilation optimisation helps abating this effect and also compensates for unnecessarily declared values.

The performance of the benchmarks can be improved further as some reduction techniques, described in [3], are not implemented yet.

Furthermore, as the number of learnable facts for secrecy check always becomes 0, the set *LearnableFact* can be set to the empty set directly without doing any compilation optimisation.

6 Discussion

6.1 Future Work

The implementation of the unbounded model in this work already allows the testing of diverse security protocols. But this implementation is only the foundation for several possible extensions, that will allow the testing of more advanced protocols, improve the usability and make the checking even more efficient.

Unifying the Checks

Currently, it is necessary to maintain different Casper input files for different security checks, i.e. one for the secrecy analysis, one for non-injective and one for injective authentication.

This is a distraction for the user, who might want to conduct a complete analysis on a protocol with one pass.

For this reason, the definition of the actual agents in the *#System* section of the Casper input should be extended. The keywords *secrecy*, *noninjectiveauth*, and *injectiveauth* could be used to designate agents for the usage in certain checks.

Casper will then have to build different systems for every check, using only designated agents.

Auto-Creation of Values and Agents

The number of values and agent configurations for different standard checks are well known, as outlined in Section 3.1.6. Therefore it is possible to create this part of the Casper input automatically, thus making the use of Casper easier.

This feature should be user-configurable.

Renaming Improvements

In Section 2.7.3 of [3], more compilation optimisations are proposed. The recommendations of the paragraph "Further Reductions of the Event Space" were not implemented yet.

The proposed compilation optimisation applies to the renaming done for the authentication checks. Reducing the number of necessary renamings will increase the performance.

The number of renamings to the commit-signal can be restricted to messages, in which agents generate only their own values, because these are the values that they want to commit to.

Furthermore, the renaming can be restricted to messages from the set `LearnableFact` (instead `KnowableFact`), as these are the only messages that the intruder is interested in.

Support for the Percentage Notation

Casper supports more features in security protocols than the ones that were addressed in this dissertation.

One of them is the "percentage"-notation, that allows to use message-parts that are not decryptable by the receiver of a message. These message parts can be called "forwarded messages".

The following messages are from a variant of the Otway-Rees protocol:

Message 1. $a \rightarrow b : a, b, na, \{na, a, b, ns\}_{ServerKey(a)}$

The receiver b is of course not able to decrypt the last part of the message.

This message is defined in the Casper input as:

1. $a \rightarrow b : a, b, na, \{na, a, b, ns\}\{ServerKey(a)\} \% v$

The difficulty of the implementation in CSP is, that the receiver b must accept a *Garbage* value or the correct message part, as he has no means to distinguish real from false values. This would allow the intruder to write messages, even if he does not know any $\{na, a, b, ns\}_{ServerKey(a)}$.

It is often possible to rewrite the protocol description to avoid the percentage notation, but it is a convenient for the user to be able to model protocols as they are.

Implementing the percentage notation makes it necessary to extend the deductions, so that they always contain the any variant of the encrypted message (e.g. $\{na, a, b, ns\}_{ServerKey(a)}$) as well as the atomic *Garbage* value. The variables within the encrypted messages do not need to have the same values as the variables in the rest of the message, even if they have the same name. Furthermore, the Tag-part of the sent-construct needs to contain non-atomic variables with the value of this forwarded message.

Conveying the Attack

A significant part of an attack on a security protocol is done as internal deduction within the intruder. The infer-events are hidden from the user and are in itself not very meaningful. This means, that the user does not know at all, how an attack works, he gets no trace of an actual run. For example, the trace of an secrecy check contains always just the leak event.

The important information that are not conveyed in these deductions are:

- Which session is this? Sessions are not very meaningful, when deductions are used to simulate unbounded parallel instances of the same protocol. A solution might be to include a reference to other infer-events or facts, on which a specific infer-event is based. The association with sessions can then made after the trace was found.

- Which message is this? This can be easily solved by adding a message-id to every deduction and infer-event.

The correct conveying of the attack trace is even more obstructed by the `Close()` function, that calculates a considerable amount of facts before the protocol is checked. Therefore, there is never any event in the trace, that explains how these fact were inferred.

However it should be possible to explain these in terms, for example, of traces in a more basic model. This would help to clarify the attacks found by the new unbounded model, where at present the intruder's ability to generate some message is not explained.

Compilation Optimisation for the Secrecy Analysis

Furthermore, the compilation optimisation is not necessary for the static secrecy analysis. As the number of learnable facts will be zero in any case, the set *LearnableFact* can be set to the empty set without any calculation.

Removal of the Sent Constructor for Certain Facts

The sent construct is used to restrict the behaviour of the internal agents. This is obviously not necessary for the last message of the agent within the protocol. Therefore, the conclusion of the last message of an agent within the protocol can be modelled without the sent constructor, thus decreasing the size of *LearnableFact*.

The difficulty in removing these sent facts lays in the renaming of the internal infer events, that still have to work.

Further Work

Currently completely unconsidered are protocols involving timestamps, algebra, and conditions.

6.2 Conclusion

This work implemented the unbounded model in Casper, allowing users to prove the absence of attacks upon security protocols under the Dolev-Yao assumption.

This implemented model simulates infinite parallel instances of the protocol and can decrease the number of checked states considerably. Benchmark results have shown, that this holds for several well tested protocol.

The drawback of the increased complexity of the calculation could be abated by numerous optimisations, that were only made possible by the internalisation method in the first place. This optimisation method has been shown to be more successful for correct protocols than for incorrect ones.

The benchmarks have demonstrated enormous improvements over the original Data Independent model of [6]. Often, ironically particularly for correct protocols, our methods show an improvement over the bounded and non-proving original Casper model.

A Casper Input rcripts

A.1 Bounded Model Yahalom Protocol Specification

```

1 #Free variables
2 a, b : Agent
3 s : Server
4 na, nb : Nonce
5 kab : SessionKey
6 ServerKey : Agent -> ServerKeys
7 InverseKeys = (kab, kab), (ServerKey, ServerKey)
8
9 #Processes
10 INITIATOR(a,na) knows ServerKey(a)
11 RESPONDER(b,s,nb) knows ServerKey(b)
12 SERVER(s,kab) knows ServerKey
13
14 #Protocol description
15 0.   -> a : b
16 1.  a -> b : na
17 2.  b -> s : a, na, nbServerKey(b)
18 3a. s -> a : b, kab, na, nbServerKey(a)
19 3b. s -> b : a, kabServerKey(b)
20 4.  a -> b : nbkab
21
22 #Specification
23 Secret(a, kab, [b,s])
24 Secret(b, kab, [a,s])
25 Secret(b, nb, [a,s])
26
27 #Actual variables
28 Alice, Bob, Ivo : Agent
29 Sam : Server
30
31 Kab : SessionKey
32 Na,Nb,Ni : Nonce
33
34 InverseKeys = (Kab, Kab)
35
36 #Inline functions
37 symbolic ServerKey
38
39 #System
40 INITIATOR(Alice, Na)
41 RESPONDER(Bob, Sam, Nb)

```

```

42 SERVER(Sam, Kab)
43
44 #Intruder Information
45 Intruder = Ivo
46 IntruderKnowledge = Alice, Bob, Ivo, Sam, Ni, ServerKey(Ivo)

```

A.2 Extended Yahalom Protocol

```

1 #Free variables
2 a, b : Agent
3 s : Server
4 na, nb : Nonce
5 kab : SessionKey
6 ServerKey : Agent -> ServerKeys
7 InverseKeys = (kab, kab), (ServerKey, ServerKey)
8
9 #Processes
10 INITIATOR(a,na) knows ServerKey(a) generates na
11 RESPONDER(b,s,nb) knows ServerKey(b) generates nb
12 SERVER(s,kab) knows ServerKey generates kab
13
14 #Protocol description
15 0. -> a : b
16 1. a -> b : na
17 2. b -> s : a, na, nbServerKey(b)
18 3a. s -> a : b, kab, na, nbServerKey(a)
19 3b. s -> b : a, kabServerKey(b)
20 4. a -> b : nbkab
21
22 #Specification
23 Secret(a, kab, [b,s])
24 Agreement(a, b, [na,nb])
25
26 #Actual variables
27 Alice, Ivo : Agent
28 Sam : Server
29
30 Kabp : SessionKey(InternalKnown)
31 Kabs : SessionKey(InternalUnknown)
32
33 Nb : Nonce(External)
34 Np : Nonce(InternalKnown)
35 Ns : Nonce(InternalUnknown)
36 Nali : Nonce(Internal)
37

```

```

38 InverseKeys = (Kabs, Kabs), (Kabp,Kabp)
39
40 #Inline functions
41 symbolic ServerKey
42
43 #System
44 INITIATOR(Alice, Np) internal
45 INITIATOR(Alice, Nali) internal
46
47 RESPONDER(Alice, Sam, Ns) internal
48 RESPONDER(Alice, Sam, Nb)
49
50 SERVER(Sam, Kabs) internal
51
52 DecideSecrecy = False
53 UnboundOptimize = True
54
55 #Intruder Information
56 Intruder = Ivo
57 IntruderKnowledge = Alice, Ivo, Sam, Np, Kabp, ServerKey(Ivo)
58 UnboundParallel = True

```

B Yahalom CSP Script

```

— *****
— *                                     Types                                     *
— *****

— Main datatype, representing all possible messages

datatype Encryption =
  Alice | Ivo | Sam | Kabp | Kabs | Np | Ns | Nb | Nali | Garbage |
  ServerKey___.Agent | Sq.Seq(Encryption) |
  Encrypt.(ALL_KEYS,Seq(Encryption)) | Hash.(HashFunction, Seq(Encryption)) |
  Xor.(Encryption, Encryption) | Sent.(Encryption, Seq(Encryption))

— All keys and hashfunctions in the system

ALL_KEYS = Union({SessionKey, ServerKeys})

ASYMMETRIC_KEYS = {k_, inverse(k_) | k_ <- ALL_KEYS, k_!=inverse(k_)}
HashFunction = {}

— All atoms in the system

ATOM = {Alice, Ivo, Sam, Kabp, Kabs, Np, Ns, Nb, Nali, Garbage}

— Some standard functions

encrypt(m_,k_) = Encrypt.(k_,m_)
decrypt(Encrypt.(k1_,m_),k_) = if k_ == inverse(k1_) then m_ else Garbage
decrypt(_,_) = Garbage
decryptable(Encrypt.(k1_,m_),k_) = k_ == inverse(k1_)

```

```

decryptable(_,_) = false
nth(ms_,n_) = if n_ == 1 then head(ms_) else nth(tail(ms_), n_ - 1)

nthts(ms_,n_) = if n_ == 1 then head(ms_) else nthts(tail(ms_), n_ - 1)

— add Garbage to a set that contains and encryption ,
— hash function application of Vernam encryption

addGarbage_(S_) =
  if S_ == {} then {Garbage}
  else Union({S_}, {Garbage | Encrypt._ <- S_},
             {Garbage | Hash._ <- S_},
             {Garbage | Xor._ <- S_})

— Definitions of user supplied functions

ServerKey(arg_1_) = ServerKey__.(arg_1_)

— Inverses of functions

inverse(Kabs) = Kabs
inverse(Kabp) = Kabp
inverse(ServerKey__.arg_) = ServerKey__.arg_

— Types in system

Agent = {Alice, Ivo}
Server = {Sam}
SessionKey = {Kabp, Kabs}
Nonce = {Np, Ns, Nb, Nali}
ServerKeys = {ServerKey(arg_1_) | arg_1_ <- Agent}

AllForegrounds = {}

— *****
— *                               Messages                               *
— *****

— Message labels

datatype Labels =
  Msg1 | Msg2 | Msg3a | Msg3b | Msg4 | Env0

MSG_BODY = {ALGEBRA_M::applyRenaming(m_) | (_,m_,_,_) <- SYSTEM_M::INT_MSG_INFO}

— Type of principals

ALL_PRINCIPALS = Union({Agent, Server})

HONEST = diff(ALL_PRINCIPALS, {Ivo})

— Channel declarations

INPUT_MSG = SYSTEM_M::INPUT_MSG
OUTPUT_MSG = SYSTEM_M::OUTPUT_MSG
DIRECT_MSG = SYSTEM_M::DIRECT_MSG
ENV_MSG = SYSTEM_M::ENV_MSG

channel receive: ALL_PRINCIPALS.ALL_PRINCIPALS.INPUT_MSG
channel send: ALL_PRINCIPALS.ALL_PRINCIPALS.OUTPUT_MSG
channel env : ALL_PRINCIPALS.ENV_MSG
channel error
channel start, close : HONEST.HONEST_ROLE

channel leak : addGarbage_(ALL_SECRETS_DI)

```

```

— Roles of agents

datatype ROLE = SPY_ | INITIATOR_role | RESPONDER_role | SERVER_role

HONEST_ROLE = diff (ROLE, {SPY_})

— Secrets in the protocol

ALL_SECRETS_0 = {}
ALL_SECRETS = addGarbage_ (ALGEBRA_M::applyRenamingToSet (ALL_SECRETS_0))

    ALL_SECRETS_DI = ALL_SECRETS

— Define type of signals, and declare signal channel

datatype Signal =
  Claim_Secret.ALL_PRINCIPALS.ALL_SECRETS.Set (ALL_PRINCIPALS) |
  Running1.HONEST_ROLE.ALL_PRINCIPALS.ALL_PRINCIPALS.Nonce.Nonce |
  Commit1.HONEST_ROLE.ALL_PRINCIPALS.ALL_PRINCIPALS.Nonce.Nonce |
  RunCom1.ALL_PRINCIPALS.ALL_PRINCIPALS.Nonce.Nonce.Nonce.Nonce |
  Running2.HONEST_ROLE.ALL_PRINCIPALS.ALL_PRINCIPALS.Nonce.Nonce |
  Commit2.HONEST_ROLE.ALL_PRINCIPALS.ALL_PRINCIPALS.Nonce.Nonce |
  RunCom2.ALL_PRINCIPALS.ALL_PRINCIPALS.Nonce.Nonce.Nonce.Nonce

channel signal : Signal

Fact_1 =
  Union({
    {Garbage},
    Agent,
    Server,
    Nonce,
    SessionKey,
    ServerKeys,
    {Encrypt.(ServerKey(b), <a, na, nb>) |
      a <- Agent, b <- Agent, na <- Nonce, nb <- Nonce},
    {Encrypt.(ServerKey(a), <b, kab, na, nb>) |
      a <- Agent, b <- Agent, kab <- SessionKey, na <- Nonce, nb <- Nonce},
    {Encrypt.(ServerKey(b), <a, kab>) |
      a <- Agent, b <- Agent, kab <- SessionKey},
    {Encrypt.(kab, <nb>) |
      kab <- SessionKey, nb <- Nonce},
    { sm | (sm,_) <- INTRUDER_M::Internal_External_Deductions}
  })

external relational_inverse_image
external relational_image
transparent chase

— *****
— *                               Honest Agents                               *
— *****

module SYSTEM_M

— Environmental messages

ENV_INT_MSG0 =
  {(Env0, b, <>) |
   b <- Agent}

ENV_MSG0 = {ALGEBRA_M::rmb(m_) | m_ <- ENV_INT_MSG0}

ENV_INT_MSG = ENV_INT_MSG0

```

```

— information about messages sent and received by agents, including
— extras fields for both agents

INT_MSG_INFO1_0 =
  {(Msg1, na, <>, <>) |
   na <- Nonce}
INT_MSG_INFO1 =
  {(Msg1, m, s, r) |
   (Msg1, m, s, r) <- INT_MSG_INFO1_0, member(m,INTRUDER_M::KnowableFact)}
INT_MSG_INFO2_0 =
  {(Msg2, Encrypt.(ServerKey(b), <a, na, nb>), <a, na, nb>, <>) |
   a <- Agent, na <- Nonce, nb <- Nonce, b <- Agent}
INT_MSG_INFO2 =
  {(Msg2, m, s, r) |
   (Msg2, m, s, r) <- INT_MSG_INFO2_0, member(m,INTRUDER_M::KnowableFact)}
INT_MSG_INFO3a_0 =
  {(Msg3a, Encrypt.(ServerKey(a), <b, kab, na, nb>), <>, <>) |
   a <- Agent, b <- Agent, kab <- SessionKey, na <- Nonce, nb <- Nonce}
INT_MSG_INFO3a =
  {(Msg3a, m, s, r) |
   (Msg3a, m, s, r) <- INT_MSG_INFO3a_0, member(m,INTRUDER_M::KnowableFact)}
INT_MSG_INFO3b_0 =
  {(Msg3b, Encrypt.(ServerKey(b), <a, kab>), <>, <>) |
   a <- Agent, b <- Agent, kab <- SessionKey}
INT_MSG_INFO3b =
  {(Msg3b, m, s, r) |
   (Msg3b, m, s, r) <- INT_MSG_INFO3b_0, member(m,INTRUDER_M::KnowableFact)}
INT_MSG_INFO4_0 =
  {(Msg4, Encrypt.(kab, <nb>), <na, nb>, <na_X_, nb>) |
   na <- Nonce, nb <- Nonce, na_X_ <- Nonce, kab <- SessionKey}
INT_MSG_INFO4 =
  {(Msg4, m, s, r) |
   (Msg4, m, s, r) <- INT_MSG_INFO4_0, member(m,INTRUDER_M::KnowableFact)}

— types of messages sent and received by agents, as they are
— considered by those agents

input_proj((l_,m_,se_,re_)) = (l_,m_,re_)
rmb_input_proj((l_,m_,se_,re_)) = ALGEBRA_M::rmb((l_,m_,re_))
output_proj((l_,m_,se_,re_)) = (l_,m_,se_)

INPUT_INT_MSG1 = { input_proj(mt_) | mt_ <- INT_MSG_INFO1 }
INPUT_INT_MSG2 = { input_proj(mt_) | mt_ <- INT_MSG_INFO2 }
INPUT_INT_MSG3a = { input_proj(mt_) | mt_ <- INT_MSG_INFO3a }
INPUT_INT_MSG3b = { input_proj(mt_) | mt_ <- INT_MSG_INFO3b }
INPUT_INT_MSG4 = { input_proj(mt_) | mt_ <- INT_MSG_INFO4 }

INPUT_INT_MSG =
  Union({
    INPUT_INT_MSG1,
    INPUT_INT_MSG2,
    INPUT_INT_MSG3a,
    INPUT_INT_MSG3b,
    INPUT_INT_MSG4
  })

OUTPUT_INT_MSG1 = { output_proj(mt_) | mt_ <- INT_MSG_INFO1 }
OUTPUT_INT_MSG2 = { output_proj(mt_) | mt_ <- INT_MSG_INFO2 }
OUTPUT_INT_MSG3a = { output_proj(mt_) | mt_ <- INT_MSG_INFO3a }
OUTPUT_INT_MSG3b = { output_proj(mt_) | mt_ <- INT_MSG_INFO3b }
OUTPUT_INT_MSG4 = { output_proj(mt_) | mt_ <- INT_MSG_INFO4 }

OUTPUT_INT_MSG =

```

```

Union({
  OUTPUT_INT_MSG1,
  OUTPUT_INT_MSG2,
  OUTPUT_INT_MSG3a,
  OUTPUT_INT_MSG3b,
  OUTPUT_INT_MSG4
})

```

— INITIATOR

```

INITIATOR_0(a, na) =
  [] b : Agent, INTRUDER_M::honest(b) @
  true and member((Env0, b,<>), ENV_INT_MSG0) &
  env_I.a.(Env0, b,<>) ->
  true and member((Msg1, na,<>), OUTPUT_INT_MSG1) &
  output.a.b.(Msg1, na,<>) ->
  [] kab : SessionKey @ [] nb : Nonce @
  [] s : Server, INTRUDER_M::honest(s) @
  true and
  member((Msg3a, Encrypt.(ServerKey(a), <b, kab, na, nb>,<>), INPUT_INT_MSG3a) &
  input.s.a.(Msg3a, Encrypt.(ServerKey(a), <b, kab, na, nb>,<>) ->
  true and member((Msg4, Encrypt.(kab, <nb>,<na, nb>), OUTPUT_INT_MSG4) &
  output.a.b.(Msg4, Encrypt.(kab, <nb>,<na, nb>) ->
  SKIP

```

```

INITIATOR_1(a, na) = INITIATOR_0(a, na)

```

```

INITIATOR(a, na) =
  INITIATOR_1(a, na)
  [[ input.s.a.(l_,m_,re_) <- receive.s.a.ALGEBRA_M::rmb((l_,m_,re_)) |
    s <- Server, (l_,m_,se_,re_) <- INT_MSG_INFO3a]]
  [[ output.a.b.(l_,m_,se_) <- send.a.b.ALGEBRA_M::rmb((l_,m_,se_)) |
    b <- Agent, (l_,m_,se_,re_) <- INT_MSG_INFO1]]
  [[ output.a.b.(l_,m_,se_) <- send.a.b.ALGEBRA_M::rmb((l_,m_,se_)) |
    b <- Agent, (l_,m_,se_,re_) <- INT_MSG_INFO4]]
  [[ env_I.a.m_ <- env.a.ALGEBRA_M::rmb(m_) |
    m_ <- ENV_INT_MSG0]]

```

— RESPONDER

```

RESPONDER_0(b, s, nb) =
  [] a : Agent, INTRUDER_M::honest(a) @ [] na : Nonce @
  true and member((Msg1, na,<>), INPUT_INT_MSG1) &
  input.a.b.(Msg1, na,<>) ->
  true and
  member((Msg2, Encrypt.(ServerKey(b), <a, na, nb>,<a, na, nb>), OUTPUT_INT_MSG2) &
  output.b.s.(Msg2, Encrypt.(ServerKey(b), <a, na, nb>,<a, na, nb>) ->
  [] kab : SessionKey @
  true and member((Msg3b, Encrypt.(ServerKey(b), <a, kab>,<>), INPUT_INT_MSG3b) &
  input.s.b.(Msg3b, Encrypt.(ServerKey(b), <a, kab>,<>) ->
  input.a.b.(Msg4, Encrypt.(inverse(kab), <nb>,<na, nb>) ->
  SKIP

```

```

RESPONDER_1(b, s, nb) = RESPONDER_0(b, s, nb)

```

```

RESPONDER(b, s, nb) =
  RESPONDER_1(b, s, nb)
  [[ input.a.b.(l_,m_,re_) <- receive.a.b.ALGEBRA_M::rmb((l_,m_,re_)) |
    a <- Agent, (l_,m_,se_,re_) <- INT_MSG_INFO1]]
  [[ input.s.b.(l_,m_,re_) <- receive.s.b.ALGEBRA_M::rmb((l_,m_,re_)) |
    s <- Server, (l_,m_,se_,re_) <- INT_MSG_INFO3b]]
  [[ input.a.b.(l_,m_,re_) <- receive.a.b.ALGEBRA_M::rmb((l_,m_,re_)) |
    a <- Agent, (l_,m_,se_,re_) <- INT_MSG_INFO4]]
  [[ output.b.s.(l_,m_,se_) <- send.b.s.ALGEBRA_M::rmb((l_,m_,se_)) |
    s <- Server, (l_,m_,se_,re_) <- INT_MSG_INFO2]]

```

— SERVER

```
SERVER_0(s, kab) =
  [] a : Agent, INTRUDER_M::honest(a) @
  [] b : Agent, INTRUDER_M::honest(b) @ [] na : Nonce @ [] nb : Nonce @
  true and member((Msg2, Encrypt.(ServerKey(b), <a, na, nb>),<>), INPUT_INT_MSG2) &
  input.b.s.(Msg2, Encrypt.(ServerKey(b), <a, na, nb>),<>) ->
  true and
  member((Msg3a, Encrypt.(ServerKey(a), <b, kab, na, nb>),<>), OUTPUT_INT_MSG3a) &
  output.s.a.(Msg3a, Encrypt.(ServerKey(a), <b, kab, na, nb>),<>) ->
  true and member((Msg3b, Encrypt.(ServerKey(b), <a, kab>),<>), OUTPUT_INT_MSG3b) &
  output.s.b.(Msg3b, Encrypt.(ServerKey(b), <a, kab>),<>) ->
  SKIP
```

SERVER_1(s, kab) = SERVER_0(s, kab)

```
SERVER(s, kab) =
  SERVER_1(s, kab)
  [[ input.b.s.(l_,m_,re_) <- receive.b.s.ALGEBRA_M::rmb((l_,m_,re_)) |
    b <- Agent, (l_,m_,se_,re_) <- INT_MSG_INFO2]]
  [[ output.s.a.(l_,m_,se_) <- send.s.a.ALGEBRA_M::rmb((l_,m_,se_)) |
    a <- Agent, (l_,m_,se_,re_) <- INT_MSG_INFO3a]]
  [[ output.s.b.(l_,m_,se_) <- send.s.b.ALGEBRA_M::rmb((l_,m_,se_)) |
    b <- Agent, (l_,m_,se_,re_) <- INT_MSG_INFO3b]]
```

— Messages as they appear on the network; each messages is renamed
 — (by rmb) to the representative member of its equivalence class

```
INPUT_MSG1 = {ALGEBRA_M::rmb(m_) | m_ <- INPUT_INT_MSG1}
INPUT_MSG2 = {ALGEBRA_M::rmb(m_) | m_ <- INPUT_INT_MSG2}
INPUT_MSG3a = {ALGEBRA_M::rmb(m_) | m_ <- INPUT_INT_MSG3a}
INPUT_MSG3b = {ALGEBRA_M::rmb(m_) | m_ <- INPUT_INT_MSG3b}
INPUT_MSG4 = {ALGEBRA_M::rmb(m_) | m_ <- INPUT_INT_MSG4}
```

```
OUTPUT_MSG1 = {ALGEBRA_M::rmb(m_) | m_ <- OUTPUT_INT_MSG1}
OUTPUT_MSG2 = {ALGEBRA_M::rmb(m_) | m_ <- OUTPUT_INT_MSG2}
OUTPUT_MSG3a = {ALGEBRA_M::rmb(m_) | m_ <- OUTPUT_INT_MSG3a}
OUTPUT_MSG3b = {ALGEBRA_M::rmb(m_) | m_ <- OUTPUT_INT_MSG3b}
OUTPUT_MSG4 = {ALGEBRA_M::rmb(m_) | m_ <- OUTPUT_INT_MSG4}
```

```
DIRECT_MSG1 = {ALGEBRA_M::rmb4(m_) | m_ <- INT_MSG_INFO1}
DIRECT_MSG2 = {ALGEBRA_M::rmb4(m_) | m_ <- INT_MSG_INFO2}
DIRECT_MSG3a = {ALGEBRA_M::rmb4(m_) | m_ <- INT_MSG_INFO3a}
DIRECT_MSG3b = {ALGEBRA_M::rmb4(m_) | m_ <- INT_MSG_INFO3b}
DIRECT_MSG4 = {ALGEBRA_M::rmb4(m_) | m_ <- INT_MSG_INFO4}
```

— Process representing Alice

```
Alpha_RESPONDER_Alice =
  Union({
    {|send.Alice.A_.m_ | A_ <- ALL_PRINCIPALS, m_ <- OUTPUT_MSG2|},
    {|receive.A_.Alice.m_ | A_ <- ALL_PRINCIPALS, m_ <- INPUT_MSG1|},
    {|receive.A_.Alice.m_ | A_ <- ALL_PRINCIPALS, m_ <- INPUT_MSG3b|},
    {|receive.A_.Alice.m_ | A_ <- ALL_PRINCIPALS, m_ <- INPUT_MSG4|}
  })
```

RESPONDER_Alice = RESPONDER(Alice, Sam, Nb)

```
Alpha_Alice =
  Union({
    {|env.Alice|},
    {|send.Alice.A_, receive.A_.Alice | A_ <- ALL_PRINCIPALS|}
  })
```

```

AGENT_Alice =
  (RESPONDER_Alice [Alpha_RESPONDER_Alice || {} ] STOP)

-- Process representing Sam

Alpha_Sam =
  Union({
    {|env.Sam|},
    {|send.Sam.A_, receive.A_.Sam | A_ <- ALL_PRINCIPALS|}
  })

AGENT_Sam = STOP

exports

ENV_MSG = {ALGEBRA_M::rmb(m_) | m_ <- ENV_INT_MSG}
INT_MSG_INFO =
  Union({
    INT_MSG_INFO1,
    INT_MSG_INFO2,
    INT_MSG_INFO3a,
    INT_MSG_INFO3b,
    INT_MSG_INFO4
  })
INPUT_MSG =
  Union({
    INPUT_MSG1,
    INPUT_MSG2,
    INPUT_MSG3a,
    INPUT_MSG3b,
    INPUT_MSG4
  })
OUTPUT_MSG =
  Union({
    OUTPUT_MSG1,
    OUTPUT_MSG2,
    OUTPUT_MSG3a,
    OUTPUT_MSG3b,
    OUTPUT_MSG4
  })
DIRECT_MSG =
  Union({
    DIRECT_MSG1,
    DIRECT_MSG2,
    DIRECT_MSG3a,
    DIRECT_MSG3b,
    DIRECT_MSG4
  })

channel input : ALL_PRINCIPALS.ALL_PRINCIPALS.INPUT_INT_MSG
channel output : ALL_PRINCIPALS.ALL_PRINCIPALS.OUTPUT_INT_MSG
channel env_I : ALL_PRINCIPALS.ENV_INT_MSG

-- Complete system

SYSTEM_0 =
  (AGENT_Alice
   |||
   AGENT_Sam)

endmodule

-- *****
-- *                               Algebra                               *
-- *****

```

```

module ALGEBRA_M

  — Algebraic laws, defined as a set of pairs

  laws = {(Garbage, Garbage)}

  — Calculate transitive closure of algebraic laws, and select
  — representative member of each equivalence class

  external mtransclose
  renaming = mtransclose(laws, Fact_1)
  ren = relational_inverse_image(renaming)

  — function that renames non-sequential fact to representative member

  applyRenaming0(a_) =
    let S_ = ren(a_)
    within if card(S_)==0 then a_ else elsing(S_)

  elsing({x_}) = x_

  domain = {a_ | (_,a_) <- renaming}

exports

  — function that renames arbitrary fact to representative member

  applyRenaming(Sq.ms_) =
    if member(Sq.ms_, Fact_1) then applyRenaming0(Sq.ms_)
    else Sq.<applyRenaming0(m_) | m_ <- ms_>
  applyRenaming(a_) = applyRenaming0(a_)

  — function that renames (label, fact, extras) triples

  rmb((l_,m_,extras_)) =
    (l_, applyRenaming(m_), applyRenamingToSeq(extras_))
  rmb4((l_,m_,s_extras_,r_extras_)) =
    (l_, applyRenaming(m_), applyRenamingToSeq(s_extras_),
     applyRenamingToSeq(r_extras_))

  — lift renaming to sets and to deductions

  applyRenamingToSet(X_) =
    union({elsing(ren(a_)) | a_ <- inter(X_,domain)}, diff(X_, domain))

  applyRenamingToSeq(X_) = <applyRenaming(e_) | e_ <- X_>

  applyRenamingToDeductions(S_) =
    {(applyRenaming0(f_), applyRenamingToSet(X_)) | (f_,X_) <- S_}

endmodule

— *****
— *                               The Intruder                               *
— *****

module INTRUDER_M

  Generations = DI_M::Generations

  — Intruder's initial knowledge

  IK0_init = {Alice, Ivo, Sam, Np, Kabp, ServerKey(Ivo), Garbage}

```

```

IK0 = IK0_init

— Unbound Parallel functions and sets, necessary for deductions

honest(x) = x != Ivo

InternalINITIATORset = { <Alice, Nali>, <Alice, Np> }
InternalRESPONDERset = { <Alice, Sam, Ns> }
InternalSERVERset = { <Sam, Kabs> }

InternalINITIATORset_external = {}
InternalRESPONDERset_external = {<Alice, Sam, Nb> }
InternalSERVERset_external = {}

— Intruder's deductions

unSq_(Sq.ms_) = set(ms_)
unSq_(m_) = {m_}

unknown_(S_) = diff(S_, IK0_init)

Base_Deductions =
  Union({SqDeductions, UnSqDeductions,
        EncryptionDeductions, DecryptionDeductions,
        VernEncDeductions, VernDecDeductions,
        FnAppDeductions, HashDeductions, SentDeductions,
        All_Internal_Deductions, UserDeductions})

SqDeductions =
  {(Sq.fs_, unknown_(set(fs_))) | Sq.fs_ <- Fact_1}

UnSqDeductions =
  {(f_, unknown_({Sq.fs_})) | Sq.fs_ <- Fact_1, f_ <- unknown_(set(fs_))}

EncryptionDeductions =
  {(Encrypt.(k_,fs_), unknown_(union({k_}, set(fs_)))) |
   Encrypt.(k_,fs_) <- Fact_1}

DecryptionDeductions =
  {(f_, unknown_({Encrypt.(k_,fs_), inverse(k_)})) |
   Encrypt.(k_,fs_) <- Fact_1, f_ <- unknown_(set(fs_))}

VernEncDeductions =
  {(Xor.(m1_,m2_), unknown_(union(unSq_(m1_), unSq_(m2_)))) |
   Xor.(m1_,m2_) <- Fact_1}

VernDecDeductions =
  {(m1l_, union(unknown_(unSq_(m2_)), {Xor.(m1_,m2_)})) |
   Xor.(m1_,m2_) <- Fact_1, m1l_ <- unSq_(m1_)}

HashDeductions = {(Hash.(f_, ms_), set(ms_)) | Hash.(f_, ms_) <- Fact_1}

— Unbound Parallel Deductions

SentDeductions = {(m_, {Sent.(m_,fs_)}) | Sent.(m_,fs_) <- Fact_1}

deductions_INITIATOR_0(a, na, b, s, nb, kab) =
  { ( Sent.(na, <a, b, na>), { b } ),
    ( Sent.(Encrypt.(kab, <nb>), <a, b, na, kab, nb>),

```

```

    { Encrypt.(ServerKey(a), <b, kab, na, nb>), Sent.(na, <a, b, na>) }
  )
}

deductions_INITIATOR_external_0(a, na, b, s, nb, kab) =
{ ( na , { b } ),
  ( Encrypt.(kab, <nb>) , { Encrypt.(ServerKey(a), <b, kab, na, nb>), na } )
}

deductions_INITIATOR_with_honest(InternalINITIATORset) =
Union({ deductions_INITIATOR_external_0(a, na, b, s, nb, kab) |
  b <- Agent, s <- Server, nb <- Nonce, kab <- SessionKey,
  <a, na> <- InternalINITIATORset, honest(a) and honest(b) and honest(s)
})

deductions_INITIATOR_external_with_honest(InternalINITIATORset) =
Union({ deductions_INITIATOR_external_0(a, na, b, s, nb, kab) |
  b <- Agent, s <- Server, nb <- Nonce, kab <- SessionKey,
  <a, na> <- InternalINITIATORset, honest(a) and honest(b) and honest(s)
})

deductions_INITIATOR_mapped(InternalINITIATORset) =
Union({ deductions_INITIATOR_external_0(a, Np, b, s, nb, kab) |
  b <- Agent, s <- Server, nb <- Nonce, kab <- SessionKey,
  <a, _> <- InternalINITIATORset, not (honest(a) and honest(b) and honest(s))
})

deductions_RESPONDER_0(a, na, b, s, nb, kab) =
{ ( Sent.(Encrypt.(ServerKey(b), <a, na, nb>), <b, s, na, a, nb>) , { na } ) }

deductions_RESPONDER_external_0(a, na, b, s, nb, kab) =
{ ( Encrypt.(ServerKey(b), <a, na, nb>) , { na } ) }

deductions_RESPONDER_with_honest(InternalRESPONDERset) =
Union({ deductions_RESPONDER_0(a, na, b, s, nb, kab) |
  a <- Agent, na <- Nonce, kab <- SessionKey,
  <b, s, nb> <- InternalRESPONDERset, honest(a) and honest(b) and honest(s)
})

deductions_RESPONDER_external_with_honest(InternalRESPONDERset) =
Union({ deductions_RESPONDER_external_0(a, na, b, s, nb, kab) |
  a <- Agent, na <- Nonce, kab <- SessionKey,
  <b, s, nb> <- InternalRESPONDERset, honest(a) and honest(b) and honest(s)
})

deductions_RESPONDER_mapped(InternalRESPONDERset) =
Union({ deductions_RESPONDER_0(a, na, b, s, Np, kab) |
  a <- Agent, na <- Nonce, kab <- SessionKey,
  <b, s, _> <- InternalRESPONDERset, not (honest(a) and honest(b) and honest(s))
})

deductions_SERVER_0(a, na, b, s, nb, kab) =
{ ( Sent.(Encrypt.(ServerKey(a), <b, kab, na, nb>), <s, a, na, nb, b, kab>) ,
  { Encrypt.(ServerKey(b), <a, na, nb>) } ),
  ( Sent.(Encrypt.(ServerKey(b), <a, kab>), <s, b, a, na, nb, kab>) ,
  { Sent.(Encrypt.(ServerKey(a), <b, kab, na, nb>), <s, a, na, nb, b, kab>) } )
}

deductions_SERVER_external_0(a, na, b, s, nb, kab) =
{ ( Encrypt.(ServerKey(a), <b, kab, na, nb>) ,
  { Encrypt.(ServerKey(b), <a, na, nb>) } ),
  ( Encrypt.(ServerKey(b), <a, kab>) , { Encrypt.(ServerKey(a), <b, kab, na, nb>) } )
}

deductions_SERVER_with_honest(InternalSERVERset) =

```

```

Union({ deductions_SERVER_0(a, na, b, s, nb, kab) |
  a <- Agent, na <- Nonce, b <- Agent, nb <- Nonce,
  <s, kab> <- InternalSERVERset, honest(a) and honest(b) and honest(s)
})

deductions_SERVER_external_with_honest(InternalSERVERset) =
  Union({ deductions_SERVER_external_0(a, na, b, s, nb, kab) |
    a <- Agent, na <- Nonce, b <- Agent, nb <- Nonce,
    <s, kab> <- InternalSERVERset, honest(a) and honest(b) and honest(s)
  })

deductions_SERVER_mapped(InternalSERVERset) =
  Union({ deductions_SERVER_0(a, na, b, s, nb, Kabp) |
    a <- Agent, na <- Nonce, b <- Agent, nb <- Nonce,
    <s, > <- InternalSERVERset, not (honest(a) and honest(b) and honest(s))
  })

Internal_Honest_Deductions =
  Union({
    deductions_INITIATOR_with_honest(InternalINITIATORset),
    deductions_RESPONDER_with_honest(InternalRESPONDERset),
    deductions_SERVER_with_honest(InternalSERVERset)
  })

All_Internal_Deductions =
  Union({
    Internal_Honest_Deductions,
    deductions_INITIATOR_mapped(InternalINITIATORset),
    deductions_RESPONDER_mapped(InternalRESPONDERset),
    deductions_SERVER_mapped(InternalSERVERset)
  })

Internal_External_Deductions =
  Union({
    All_Internal_Deductions,
    deductions_INITIATOR_external_with_honest(InternalINITIATORset_external),
    deductions_RESPONDER_external_with_honest(InternalRESPONDERset_external),
    deductions_SERVER_external_with_honest(InternalSERVERset_external)
  })

All_Internal_Honest_Deductions =
  union(Internal_Honest_Deductions, Incomplete_Base_Deductions)

All_Internal_External_Deductions =
  union(Internal_External_Deductions, Incomplete_Base_Deductions)

Incomplete_Base_Deductions =
  Union({ SqDeductions, UnSqDeductions,
    EncryptionDeductions, DecryptionDeductions,
    VernEncDeductions, VernDecDeductions,
    FnAppDeductions, HashDeductions, SentDeductions, UserDeductions })

UserDeductions = {}

FnAppDeductions = {}

— close up intruder's initial knowledge under deductions;
— calculate which facts cannot be learnt

components_(Sq.ms_) =
  if member(Sq.ms_, Fact_1) then {Sq.ms_} else set(ms_)
components_(m_) = {m_}

```

```

Seeable_ =
  Union({unknown_(components_(m_)) | (_,m_,_,_) <- SYSTEM_M::INT_MSG_INFO})

Close_(IK_, ded_, fact_) =
  let IK1_ =
    union(IK_, {f_ | (f_,fs_) <- ded_, fs_ <= IK_})
    ded1_ =
      {(f_,fs_) | (f_,fs_) <- ded_, not (member(f_,IK_)),
              fs_ <= fact_}
  within
  if card(IK_)==card(IK1_) and card(ded_)==card(ded1_)
  then (IK_, {(f_, diff(fs_,IK_)) | (f_,fs_) <- ded_})
  else Close_(IK1_, ded1_, fact_)

(KnowledgeFact, _) =
  Close_(ALGEBRA_M::applyRenamingToSet(IK0),
        ALGEBRA_M::applyRenamingToDeductions(All_Internal_External_Deductions),
        ALGEBRA_M::applyRenamingToSet(Fact_1))

(IK1, Deductions) =
  Close_(ALGEBRA_M::applyRenamingToSet(IK0),
        ALGEBRA_M::applyRenamingToDeductions(Base_Deductions),
        ALGEBRA_M::applyRenamingToSet(KnowledgeFact))

LearnableFact = diff(KnowledgeFact, IK1)

Deductions' = — Don't you hate hacks like this?
  if Deductions=={} then {(Garbage,{Garbage})} else Deductions

— The intruder

— * leak is used to signal that a possible secret has been learnt
— * hear and say are used to represent hearing or saying a message
— * infer(f,fs) represent deducing fact f from the set of facts fs

— Types of sender and receiver of each message

SenderType (Msg1) = Agent
SenderType (Msg2) = Agent
SenderType (Msg3a) = Server
SenderType (Msg3b) = Server
SenderType (Msg4) = Agent

ReceiverType(Msg1) = Agent
ReceiverType(Msg2) = Server
ReceiverType(Msg3a) = Agent
ReceiverType(Msg3b) = Agent
ReceiverType(Msg4) = Agent

— Component of intruder for currently unknown fact f_ :
— * ms_ is the set of messages that contain f_ at the top level
— * fss_ is the set of sets of facts from which f_ can be deduced
— * ds_ is the set of deductions that use f_

IGNORANT(f_,ms_,fss_,ds_) =
  hear?m_:ms_ -> KNOWS(f_,ms_,ds_)
  []
  ([| fs_ : fss_, not (member(f_,fs_)) |] @
   infer.(f_,fs_) -> KNOWS(f_,ms_,ds_))

— Component of intruder for known fact f_

KNOWS(f_,ms_,ds_) =
  hear?m_:ms_ -> KNOWS(f_,ms_,ds_)

```

```

[]
say?m_:ms_ -> KNOWS(f_,ms_,ds_)
[]
([| ded@@(f1_,fs_) : ds_, f1_!=f_ @ infer.ded -> KNOWS(f_,ms_,ds_)
[]
member(f_,ALL_SECRETS_DI) & leak.f_ -> KNOWS(f_,ms_,ds_)

— Alphabet of this component

AlphaL(f_,ms_,fss_,ds_) =
  Union({(if member(f_,ALL_SECRETS_DI) then {leak.f_} else {}),
        {hear.m_, say.m_ | m_ <- ms_},
        {infer.(f_,fs_) | fs_ <- fss_},
        {infer.(f1_,fs_) | (f1_,fs_) <- ds_}
        })

— Set of all (f_, ms_, fss_, ds_) for which intruder components
— must be built

f_ms_fss_ds_s =
  let rid_ = relational_image(Deductions)
      msf_ = relational_image({(f_, m_) | m_ <- MSG_BODY, f_ <- unSq_(m_)})
      xsf_ = relational_image({(f_, x_) | x_@@(f_,fs_) <- Deductions,
                                f_ <- fs_})
  within {(f_, msf_(f_), rid_(f_), xsf_(f_)) | f_ <- LearnableFact}

— Put components together in parallel ...

INTRUDER_00 =
  ([| (f_,ms_,fss_,ds_) : f_ms_fss_ds_s @
    [AlphaL(f_,ms_,fss_,ds_)] IGNORANT(f_,ms_,fss_,ds_))

INTRUDER_0 = INTRUDER_00 \ {|infer|}

INTRUDER_RUNNING_0_1 =
  let Agent_renamed_ = ALGEBRA_M::applyRenamingToSet(Agent)
      SessionKey_renamed_ = ALGEBRA_M::applyRenamingToSet(SessionKey)
      Nonce_renamed_ = ALGEBRA_M::applyRenamingToSet(Nonce)
  within
  INTRUDER_00 [| infer.(Sent.(Encrypt.(kab, <nb>), <a, b, na, kab, nb>),fs_) <-
    signal.Running1.INITIATOR_role.a.b.na.nb |
    a <- Agent_renamed_, b <- Agent_renamed_,
    kab <- SessionKey_renamed_, na <- Nonce_renamed_,
    nb <- Nonce_renamed_, (f_,fs_) <- Deductions,
    member((Sent.(Encrypt.(kab, <nb>), <a, b, na, kab, nb>),fs_),Deductions)
  ] \ {|infer|}

INTRUDER_RUNNING_0_2 =
  let Agent_renamed_ = ALGEBRA_M::applyRenamingToSet(Agent)
      Nonce_renamed_ = ALGEBRA_M::applyRenamingToSet(Nonce)
      Server_renamed_ = ALGEBRA_M::applyRenamingToSet(Server)
  within
  INTRUDER_00 [|
    infer.(Sent.(Encrypt.(ServerKey(b), <a, na, nb>), <b, s, na, a, nb>),fs_) <-
    signal.Running2.RESPONDER_role.b.a.na.nb |
    a <- Agent_renamed_, b <- Agent_renamed_, na <- Nonce_renamed_,
    nb <- Nonce_renamed_, s <- Server_renamed_, (f_,fs_) <- Deductions,
    member((Sent.(Encrypt.(ServerKey(b), <a, na, nb>), <b, s, na, a, nb>),fs_),Deductions)
  ] \ {|infer|}

— ... and rename events appropriately

INTRUDER_1(INTRUDER_0) =
  (chase(INTRUDER_0)
  [| hear.m_ <- send.A_.B_.(l_,m_,se_) |

```

```

        (l_,m_,se_,re_) <- DIRECT_MSG,
        A_ <- diff(SenderType(l_),{Ivo}), B_ <- ReceiverType(l_) ]]
[[{| hear |}] STOP)
[[ say.m_ <- receive.A_.B_.(l_,m_,re_) |
    (l_,m_,se_,re_) <- DIRECT_MSG,
    A_ <- SenderType(l_), B_ <- ReceiverType(l_) ]]

— Add in facts that are known initially

SAY_KNOWN_0 =
(inter(IK1, ALL_SECRETS_DI) != {} & dummy_leak -> SAY_KNOWN_0)
[] dummy_send -> SAY_KNOWN_0
[] dummy_receive -> SAY_KNOWN_0

SAY_KNOWN =
SAY_KNOWN_0
[[ dummy_leak <- leak.f_ | f_ <- inter(IK1, ALL_SECRETS_DI) ]]
[[ dummy_send <- send.A_.B_.(l_,m_,se_) |
    (l_,m_,se_,re_) <- DIRECT_MSG, components_(m_) <= IK1,
    A_ <- diff(SenderType(l_),{Ivo}), B_ <- ReceiverType(l_) ]]
[[ dummy_receive <- receive.A_.B_.(l_,m_,re_) |
    (l_,m_,se_,re_) <- DIRECT_MSG, components_(m_) <= IK1,
    A_ <- SenderType(l_), B_ <- ReceiverType(l_) ]]

STOP_SET = {| send.Ivo |}

exports

— Declare channels:
channel hear, say : MSG_BODY
channel infer : Deductions'
channel dummy_leak, dummy_send, dummy_receive

print IK1          — intruder's initial knowledge
print KnowableFact — all facts that might be learnt
print Deductions   — all deductions over KnowableFact

— Complete intruder

INTRUDER(INTRUDER_0) =
    (INTRUDER_1(INTRUDER_0) [| STOP_SET |] STOP) [|] SAY_KNOWN

endmodule

IntruderInterface = {| send, receive |}

SYSTEM_PARAM(INTRUDER_0) =
    SYSTEM_M::SYSTEM_0 [| IntruderInterface |] INTRUDER_M::INTRUDER(INTRUDER_0)
SYSTEM =
    SYSTEM_PARAM(INTRUDER_M::INTRUDER_0)

— *****
— *                               Specifications and Assertions                               *
— *****

— Authentication specifications

— Authentication specification number 1

module AUTH1_M

— Spec parameterized by name of agent being authenticated

AuthenticateINITIATORToRESPONDERAgreement_na_nb_0(a) =
    signal.Running1.INITIATOR_role.a?b?na?nb ->

```

```

    signal.Commit1.RESPONDER_role.b.a.na.nb -> STOP

AlphaAuthenticateINITIATORToRESPONDERAgreement_na_nb_0(a) =
  {| signal.Running1.INITIATOR_role.a.b,
    signal.Commit1.RESPONDER_role.b.a |
    b <- inter(Agent, HONEST)|}

— Specs for particular agents being authenticated

AuthenticateINITIATORALiceToRESPONDERAgreement_na_nb =
  AuthenticateINITIATORToRESPONDERAgreement_na_nb_0(Alice)
|||
AuthenticateINITIATORToRESPONDERAgreement_na_nb_0(Alice)

— alphabet of specification

alphaAuthenticateINITIATORToRESPONDERAgreement_na_nb =
  AlphaAuthenticateINITIATORToRESPONDERAgreement_na_nb_0(Alice)

exports

— Specs for all agents being authenticated

AuthenticateINITIATORToRESPONDERAgreement_na_nb =
  AuthenticateINITIATORALiceToRESPONDERAgreement_na_nb

— System for authentication checking

SYSTEM_1 =
  let Agent_renamed_ = ALGEBRA_M::applyRenamingToSet(Agent)
      Nonce_renamed_ = ALGEBRA_M::applyRenamingToSet(Nonce)
      SessionKey_renamed_ = ALGEBRA_M::applyRenamingToSet(SessionKey)
  within
  SYSTEM_PARAM(INTRUDER_M::INTRUDER_RUNNING_0_1)
  [| send.a.b.ALGEBRA_M::rmb((Msg4, Encrypt.(kab, <nb>), <na, nb>)) <-
    signal.Running1.INITIATOR_role.a.b.na.nb,
receive.a.b.ALGEBRA_M::rmb((Msg4, Encrypt.(kab, <nb>), <na, nb>)) <-
  signal.Commit1.RESPONDER_role.b.a.na.nb |
  a <- Agent_renamed_, b <- Agent_renamed_, na <- Nonce_renamed_,
  nb <- Nonce_renamed_, kab <- SessionKey_renamed_,
  member((Msg4, Encrypt.(kab, <nb>), <na, nb>),SYSTEM_M::OUTPUT_INT_MSG4),
  member((Msg4, Encrypt.(kab, <nb>), <na, nb>),SYSTEM_M::INPUT_INT_MSG4)
  ||
  \ diff(Events, alphaAuthenticateINITIATORToRESPONDERAgreement_na_nb)

endmodule

assert AUTH1_M::AuthenticateINITIATORToRESPONDERAgreement_na_nb [T=
  AUTH1_M::SYSTEM_1

— Authentication specification number 2

module AUTH2_M

— Spec parameterized by name of agent being authenticated

AuthenticateRESPONDERToINITIATORAgreement_na_nb_0(b) =
  signal.Running2.RESPONDER_role.b?a?na?nb ->
  signal.Commit2.INITIATOR_role.a.b.na.nb -> STOP

AlphaAuthenticateRESPONDERToINITIATORAgreement_na_nb_0(b) =
  {| signal.Running2.RESPONDER_role.b.a,
    signal.Commit2.INITIATOR_role.a.b |
    a <- inter(Agent, HONEST)|}

```

```

— Specs for particular agents being authenticated

AuthenticateRESPONDERAliceToINITIATORAgreement_na_nb =
  AuthenticateRESPONDERToINITIATORAgreement_na_nb_0(Alice)
|||
AuthenticateRESPONDERToINITIATORAgreement_na_nb_0(Alice)

— alphabet of specification

alphaAuthenticateRESPONDERToINITIATORAgreement_na_nb =
  AlphaAuthenticateRESPONDERToINITIATORAgreement_na_nb_0(Alice)

exports

— Specs for all agents being authenticated

AuthenticateRESPONDERToINITIATORAgreement_na_nb =
  AuthenticateRESPONDERAliceToINITIATORAgreement_na_nb

— System for authentication checking

SYSTEM_2 =
  let Agent_renamed_ = ALGEBRA_M::applyRenamingToSet(Agent)
      Server_renamed_ = ALGEBRA_M::applyRenamingToSet(Server)
      Nonce_renamed_ = ALGEBRA_M::applyRenamingToSet(Nonce)
      SessionKey_renamed_ = ALGEBRA_M::applyRenamingToSet(SessionKey)
  within
  SYSTEM_PARAM(INTRUDER_M::INTRUDER_RUNNING_0_2)
  [[ send.b.s.ALGEBRA_M::rmb((Msg2, Encrypt.(ServerKey(b), <a, na, nb>), <a, na, nb>))
    <- signal.Running2.RESPONDER_role.b.a.na.nb,
  send.a.b.ALGEBRA_M::rmb((Msg4, Encrypt.(kab, <nb>), <na, nb>)) <-
    signal.Commit2.INITIATOR_role.a.b.na.nb |
    b <- Agent_renamed_, s <- Server_renamed_, a <- Agent_renamed_,
    na <- Nonce_renamed_, nb <- Nonce_renamed_,
    kab <- SessionKey_renamed_,
    member((Msg2, Encrypt.(ServerKey(b), <a, na, nb>), <a, na, nb>),
           SYSTEM_M::OUTPUT_INT_MSG2),
    member((Msg4, Encrypt.(kab, <nb>), <na, nb>),SYSTEM_M::OUTPUT_INT_MSG4)
  ]]
  \ diff(Events, alphaAuthenticateRESPONDERToINITIATORAgreement_na_nb)

endmodule

assert AUTH2_M::AuthenticateRESPONDERToINITIATORAgreement_na_nb [T=
  AUTH2_M::SYSTEM_2

```

C Excerpt from the Casper Extension

The following Haskell function calculates the deductions from a protocol description and is the core of the extension of Casper:

```

>makeSentRep :: ProtDesc' ->      — Protocol description
>      VarTypeList ->
>      SentRep
>makeSentRep (first_message : protdesc) fvts =
>  let
>    (_,_,first_message_sender, first_message_receiver,first_message_content,_,_,_)
>    = first_message
>  in
>    makeSentRep2 protdesc
>      [(first_message_receiver,first_message_content)] []
>      [(first_message_sender,receiverSimpleAtoms first_message_content)]
>      fvts

>makeSentRep2 :: ProtDesc' ->
>      [(Player,Msg)] ->
>      [(Player,Msg)] ->
>      [(Player,[Msg])] ->
>      VarTypeList -> SentRep
>makeSentRep2 [] _ _ _ _ = []
>makeSentRep2 (msg:msglist) lastreceived_list lastsent_list initiated_knowledge fvts =
>  let
>    (al, mn, sender, receiver, msgcontent, (test,uknw), vn11, vn12) = msg
>    atoms_in_msg
>    = senderSimpleAtoms msgcontent
>    sender_received_messages
>    = [m | (pl,m) <- lastreceived_list, pl == sender]
>    sender_received_atoms
>    = concat (map receiverSimpleAtoms sender_received_messages)
>
>    sender_introduced_atoms
>    = concat [ia | (pl,ia) <- initiated_knowledge, pl == sender]
>
>    last_sent_player
>    = [m | (pl,m) <- lastsent_list, pl == sender]
>    next_initiated_player_knowledge
>    = remdups ([playerAtom sender,playerAtom receiver]
>              ++ sender_introduced_atoms
>              ++ sender_received_atoms
>              ++ atoms_in_msg)
>    next_sent_player
>    = (Sent msgcontent (remdups next_initiated_player_knowledge))
>    next_sent
>    = (sender,next_sent_player) : filter (\ (p,m) -> p /= sender) lastsent_list
>    next_received_player
>    = [(receiver, msgcontent)]
>    next_received_list
>    = filter (\ (p,m) -> p /= sender) lastreceived_list ++ next_received_player
>    next_initiated_knowledge
>    = (sender, next_initiated_player_knowledge)
>      : filter (\ (p,m) -> p /= sender) initiated_knowledge
>    test_str
>    = if (test == []) then "" else (",_" ++ test)
>    deduction
>    = (sender,next_sent_player, sender_received_messages,
>      last_sent_player, test_str,mn)
>  in
>    deduction
>    : makeSentRep2 msglist next_received_list next_sent next_initiated_knowledge fvts

```

References

- [1] P. Broadfoot. *Data independence in the model checking of security protocols*. PhD thesis, University of Oxford, September 2001.
- [2] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 1983.
- [3] E. Kleiner. *A Web Services Security study using Casper and FDR*. PhD thesis, University of Oxford, 2006.
- [4] E. Kleiner and A. W. Roscoe. Modelling unbounded parallel sessions of security protocols in csp. *Journal of Computer Security*, 2005.
- [5] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.
- [6] Roscoe. Proving security protocols with model checkers by data independence techniques. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.
- [7] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
- [8] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. Modelling and analysis of security protocols, 2001.