MODELING AND VERIFICATION OF A STREAM AUTHENTICATION PROTOCOL USING COMMUNICATING SEQUENTIAL PROCESSES

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Süleyman Murat ÖZKAN

August 2010
İZMİR

We approve the thesis of **Süleyman Murat ÖZKAN**

Prof. Dr. Sıtkı AYTAÇ
Supervisor

Assoc. Prof. Dr. Ahmet KOLTUKSUZ
Co-Supervisor

Assist. Prof. Dr. Ufuk ÇELİKKAN
Committee Member

Assist. Prof. Dr. Gökhan DALKILIÇ
Committee Member

Assist. Prof. Dr. Serap ATAY
Committee Member

26 August 2010

Prof. Dr. Sıtkı AYTAÇ
Head of the Department of
Computer Engineering

Assoc. Prof. Dr. Talat YALÇIN
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGEMENTS

First, I owe my gratitude to my co-advisor Assoc. Prof. Dr. Ahmet Koltuksuz for his guidance and patience.

I also would like to thank Prof. Dr. Sıtkı Aytaç for his support and encouragement. Additionally, I would like to thank Asst. Prof. Dr. Serap Atay for her invaluable support.

Furthermore, I would like to express my gratitude to (almost Dr.) Selma Tekir and Mutlu Beyazit. This study would not have been possible without the wisdom, guidance, friendship and humor. My roommate Gürcan Gerçek deserves my thanks too.

I also would like to thank to Dr. Wayne Trotman, for his help in correcting typos and errors in this study.

Finally, I would like to thank my family most, for their never-ending support. Especially, my fiancé and colleague Burcu Külahçıoğlu deserves my sincere thanks. This study could never be finished without her support, encouragement and love.

ABSTRACT

MODELING AND VERIFICATION OF A STREAM AUTHENTICATION PROTOCOL USING COMMUNICATING SEQUENTIAL PROCESSES

Although most systems used for computation are concurrent systems, classical theories of computation are generally involved in sequential formalisms. Thus, mathematical methods are developed for modeling and analyzing the behavior of concurrent and reactive systems. One of these formal methods is Communicating Sequential Processes (CSP), which is a process algebra proposed by Hoare in the 1970s. Broad theory of CSP captures different properties of processes by using different approaches within a unifying formalization.

Many security protocols are modeled with CSP and successfully verified using model-checking or theorem proving techniques. Unlike other authentication protocols modeled using CSP, each of the Efficient Multi-chained Stream Signature (EMSS) protocol messages are linked to the previous messages, forming hash chains, which introduce difficulties into the modeling and verification. In this thesis the EMSS stream authentication protocol is modeled using CSP and its authentication properties are verified using model checking, which in turn calls for building an infinite state model of the protocol that is also successfully reduced into a finite state model.

ÖZET

HABERLEŞEN SIRALI SÜREÇLER KULLANARAK BİR AKIŞ KİMLİK DENETİMİ PROTOKOLÜNÜN MODELLENMESİ VE DOĞRULANMASI

Hesaplamada kullanılan sistemlerin çoğu eşzamanlı sistemler olmasına rağmen, klasik hesaplama kuramı genel olarak sıralı sistemler üzerinde durmaktadır. Bu nedenle, eşzamanlı ve birbiriyle iletişim içinde olan sistemlerin modellenmesi ve doğrulanması için matematiksel yöntemler geliştirilmiştir. Bu yöntemlerden birisi olan Haberleşen Sıralı Süreçler (CSP), Hoare tarafından yetmişli yıllarda geliştirilen bir süreç cebiridir. CSP'nin kapsamlı kuramı, süreçlerin değişik özelliklerini, değişik yaklaşımları kullanarak yakalamaya izin verir.

Birçok güvenlik protokolü CSP ile modellenmiş ve model denetimi veya kuram ispatı teknikleri kullanılarak bu protokollerin doğruluğu gösterilmiştir. CSP ile modellenmiş olan diğer kimlik denetimi protokollerinden farklı olarak, her bir Verimli Çoklu-Zincirli Akış İmzası (EMSS) protokol mesajı, modelleme ve doğrulamada zorluk yaratacak şekilde, önceki mesajlara özet zincirleri ile bağlıdır. Bu tezde, EMSS akış kimlik denetimi protokolü CSP ile modellenmiş ve kimlik denetimi özellikleri, model denetimi yöntemlerini kullanarak doğrulanmıştır.

TABLE OF CONTENTS

LIST OF FIGURES ix

LIST OF TABLES x

CHAPTER 1. INTRODUCTION 1

CHAPTER 2. CONCURRENT SYSTEMS AND COMMUNICATING SEQUENTIAL PROCESSES 3

2.1. Concurrent Systems 3

2.1.1. Models of Concurrency 4

2.1.2. Formal Verification Techniques 4

2.2. Communicating Sequential Processes (CSP) 5

2.3. Events, Processes and Operations 5

2.4. Operational Semantics 7

2.5. Behaviors and Denotational Semantics 8

2.5.1. Traces ($\mathcal{T}$) Model 8

2.5.2. Stable Failures ($\mathcal{F}$) Model 9

2.5.3. Failures / Divergences ($\mathcal{N}$) Model 10

2.6. Refinement Relations 11

2.7. A Verification Tool for CSP: FDR 12

2.7.1. Refinement Checking 13

2.7.2. Compression Techniques 14

CHAPTER 3. MODELING AND VERIFICATION OF SECURITY PROTOCOLS USING CSP 15

3.1. Stream Authentication 15

3.2. The Generic CSP Protocol Model 16

3.2.1. Honest Agents 17

3.2.2. Intruder Model 17

3.2.3. Simplifications on Protocol Model 18

3.3. Data Independence 19
3.4. Specification of Security Protocols 21
3.4.1. Using $signal$ Events 21
3.4.2. Authentication 21

CHAPTER 4. PROPOSED MODEL FOR THE PROTOCOL 23
4.1. The Efficient Multi-chained Stream Signature (EMSS) Protocol 23
4.2. Modeling Assumptions 25
4.3. Symbolic Data Types and Operations 25
4.3.1. Symbolic Data Types 26
4.3.2. Cryptographic Operations 27
4.4. Modeling the Hash Chain Mechanism using Fixed Hash Sizes 27
4.4.1. Using Fixed Hash Sizes 28
4.4.2. The Construction of $DataHash$ Set 29
4.4.3. The $hash$ function 31
4.4.4. Implementation Considerations 32
4.5. Infinite State CSP Model of the EMSS Protocol 35
4.5.1. Sender Agent 35
4.5.2. Receiver Agent 36
4.5.3. Check Process 37
4.5.4. Intruder and the Network 39
4.5.5. Example Correct Run 42
4.6. Reduction of Infinite State Model to Finite Model 44
4.6.1. Analyzing the Infinite State CSP Model 44
4.6.2. Revisions on the Data Model 45
4.6.3. Bounding the Size of Hash Sets 46
4.6.4. Revisions on Agent Processes 47
4.6.5. Example Run using the Revised Model 49
4.7. Labeling the Sequence of Messages 50

CHAPTER 5. SPECIFICATION AND VERIFICATION OF THE PROPOSED PROTOCOL MODEL 51
5.1. Parameters and Configurations Used 51

5.2. Specifications of the Proposed Model ... 52
5.2.1. Validation Specifications ... 53
5.2.2. Verification Specifications ... 54
5.3. Refinement Checking of the Proposed Model ... 55

CHAPTER 6. CONCLUSIONS ... 58

REFERENCES ... 60

APPENDIX A ... 66

LIST OF FIGURES

Figure **Page**

Figure 3.1. Generic CSP Model Used for Verification of Security Protocols 16
Figure 4.1. The Hash Chains Between the Protocol Messages in EMSS 24
Figure 4.2. Each Message Must Use the Hashes of Preceding Messages 30
Figure 4.3. Actual Data Types Used in the Protocol Model 33
Figure 4.4. The PM Set Including the Signature Message 33
Figure 4.5. The Function that Computes the Cardinality of $DataHash$ 34
Figure 4.6. The $DataHash$ Set of Size 3 34
Figure 4.7. The Algorithm of $Check$ Process 38
Figure 4.8. The Network Model Used for the Protocol 41
Figure 4.9. Correct and Incorrect Hash Values 46
Figure 5.1. The Debug Trace of the First Validation Property 56
Figure 5.2. The Debug Trace for Second Property, using $IntruderFC$ 56

LIST OF TABLES

Table **Page**

Table 4.1. The Size of $DataHash$ Set for Two Data Values. 44

Table 5.1. Results of Refinement Checks of Validation Properties 55

Table 5.2. Results of Refinement Checks for Verification Properties............................. 57

CHAPTER 1

INTRODUCTION

Systems in the real world are composed of many interacting components, which can, in turn, interact with other systems, constituting a system exhibiting more complex behavior than their sub-components. Also, these systems run concurrently; that is, events may occur simultaneously instead of sequentially.

Computational systems are good examples of these systems. For example, an operating system can run multiple processes simultaneously which interact with each other. Even though this is only an illusion, the interaction of the processes with each other and the environment makes it a concurrent system, interacting and evolving in parallel with each other.

On the other hand, most classical theories of computation are involved in sequential formalisms of computation, such as automata theory and Turing machines. These formalisms consider computation as a transformation of input into output, while interaction with environment (or other systems) is left unconsidered.

The deficiencies of sequential computing models led to the formation of concurrent formalisms of computation, starting with the introduction of Petri nets (Petri 1962). Currently, there are different approaches for modeling concurrent systems based on graph (e.g. Labeled Transition Systems), net (e.g. Petri Nets) and term (e.g. Process algebras) structures, which have different levels of expressiveness. The specifications of a model can be stated in the same formalism as the model itself or in a different formalism, which are used to verify the correctness of the system (Glabbeek 2000).

Another aspect of concurrent systems is the dependence on the concept of time. Concurrency theory models time in its formalisms in two different approaches. The first aspect is the qualitative notion of time, which does not keep track of the actual units of time elapsed between events, but the relative ordering of the events. The other models use quantitative notion of time, in which the passage of time between two events can be explicitly modeled.

Communicating Sequential Processes (CSP) is essentially a term based model. The model of the system and the specifications are represented as CSP processes. Then, the correctness of the system is determined by means of refinement relations.

The contribution of this study is to model the Efficient Multi-chained Stream Signature (EMSS) protocol using CSP and verify its authentication properties using model checking. The infinite state model of the EMSS protocol is built using fixed sized hashes. This model is not suitable for verification by model checking; therefore it has been reduced into a finite state model and verified using Failures - Divergences Refinement (FDR) (Goldsmith 2005a) tool. This study uses model checking to verify the security properties of EMSS protocol, which have previously been verified only using theorem proving methods (ter Beek, et al. 2003, 2006, Perrig, et al. 2000).

In the second chapter, the theory of Communicating Sequential Processes and its place in the concurrent computing models are introduced. The fundamental operators and concepts of CSP and their role in the operational semantics of the language are presented. Additionally, behavior of the processes is discussed within the denotational semantics section. Finally, the concept of refinement is introduced and tool support for CSP is evaluated.

The purpose of the third chapter is to introduce general concepts and techniques used for modeling and verification of security protocols, as the protocol model of EMSS constructed in this thesis modifies and uses these concepts.

In the fourth chapter, the EMSS protocol is introduced and CSP model of the EMSS protocol is built. In order to do that, infinite state CSP model of the protocol is first built and then reduced to a finite state model while retaining the properties of the protocol.

The fifth chapter involves building the specification processes and performing validation and verification of the model using refinement checking on the reduced model. Additionally, the reduced model is evaluated using results of the check.

The last chapter states the conclusions and contributions of this study.

CHAPTER 2

CONCURRENT SYSTEMS AND COMMUNICATING SEQUENTIAL PROCESSES

In this chapter, the theory of Communicating Sequential Processes (CSP) and its place in the concurrent computing models are introduced. The fundamental operators and concepts of CSP and their role in the operational semantics of the language are presented. Additionally, behavior of the processes is discussed within the denotational semantics section. Finally, the concept of refinement is introduced and tool support for CSP is evaluated.

2.1. Concurrent Systems

Computational systems are mostly composed of subcomponents which interact with each other and form a more complex structure. Concurrency theory forms the mathematical foundations for representing and analyzing these systems in a systematic manner.

Concurrent systems share some common traits. First, concurrent systems have interaction. Interaction might be implemented differently in different models, but in general, it allows a system to communicate with another system.

The second trait is non-transformality. Transformal models are defined within the classical theories of computation, in which the system is only an input – output transformer, such as an automaton. However, concurrent systems interact with each other, they tend to evolve with their environments and their outputs may not entirely depend on the inputs. The final trait is unbounded execution. In sequential models, the model need to stop, but concurrent systems do not have this restriction. (Bowman and Gomez 2006).

2.1.1. Models of Concurrency

There are numerous models, which represent concurrent systems. These models can be classified with respect to the kind of mathematical objects that are used to represent processes (Winskel and Nielsen 1995).

In Graph oriented models (Keller 1976), a system is represented by a *process graph*, or *state-transition diagram*. A variant is *labelled transition systems*, in which a concurrent system is not denoted by a whole graph, but by a vertex in a graph. Another class is net oriented models, in which a concurrent system is represented by a Petri net. Event oriented models represents a concurrent system by a set of *events* (action occurrences) together with some structure on this set, determining the relations between the events (Mazurkiewicz 1988).

In term models, a system is represented as a term in a system description language, such as Calculus of Communicating Systems (CCS) (Milner 1982) or CSP. Using term models, a system can be denoted either by means of an expression (i.e. term) in that language or *semantically*.

CSP is a term based model. In CSP, processes can be viewed in three different aspects, each of which gives different meanings to processes (Roscoe 1998):

(i) *Operational Semantics* interprets CSP processes as transition diagrams and helps generating the state space of processes from process definitions.

(ii) *Denotational Semantics* interprets CSP processes in terms of the behaviors that the processes engage in. Different behaviors are represented as different models.

(iii) *Algebraic Semantics* involves in the algebraic relations between operators and defines process equivalences in terms of relations.

2.1.2. Formal Verification Techniques

After expressing the system using a model, verification results need to be established on this model. To establish verification results on a model, the requirements of the model should be formalized first.

Then, the model is verified using theorem-proving; which uses logical inference rules in order to prove or disprove the specifications. Alternatively, the model can be verified using model checking; which involves in building the state space of the system and exhaustively checking all states against the specifications.

2.2. Communicating Sequential Processes (CSP)

CSP (Hoare 1978, Roscoe 1998, Schneider 1999, Davies 2006) is a process algebra, designed specifically for the description of communication patterns of concurrent system components that interact through message passing. The original version of CSP is introduced in the seventies and further developed and refined to modern form as process algebra (Brookes et al 1984, Hoare 1985) which is widely influenced by Calculus of Communicating Systems (Milner 1982).

Machine readable CSP or CSP_M (Scattergood 1998) is a functional language with CSP constructs represented in ASCII form.

2.3. Events, Processes and Operations

In CSP, systems are modeled in terms of the *events* that they can perform. Events are atomic and have no measurable duration. Also, the intervals of time between events are not considered, only the relative ordering of events is important. Timed CSP (Reed and Roscoe, 1998) is an extension to CSP, which introduces timing information in processes, in order to model the passage of time.

A single event may contain various pieces of information, so events can have structure. If M and N are two sets of messages, then $M.N$ will be the set of messages $\{m.n \mid m \in M \wedge n \in N\}$.

Events can be associated with *channels*, which model the point to point communications between processes. A channel c is said to be of type M if any event $c.m \in \Sigma$ has that $m \in M$.

Definition (Alphabet): The set of all possible events (Σ) forms the *alphabet* of the process and denoted as Σ. An alphabet can be finite (Σ^*) or infinite (Σ^ω).

The τ event specifies an internal transition that cannot be observed from environment.

Definition (Process): *Processes* are the components of systems, which are the entities, described using CSP in terms of the possible events they may engage in.

The BNF form of a CSP process can be defined as:

$$P, Q ::= STOP \mid SKIP \mid a \rightarrow P \mid P; Q \mid P \,\square\, Q \mid P \sqcap Q \mid a: A \rightarrow P(a) \mid$$
$$P \vartriangle Q \mid f(P) \mid P \setminus A \mid P \;{}_{A}||_{B} Q \mid P \underset{A}{||} Q \mid \;||_{A_P} P \mid P \;|||\; Q \mid \mu X \cdot F(X)$$

The process $STOP$ is the process that can engage in no events at all; it is equivalent to *deadlock*. $SKIP$ denotes a process that does nothing but immediately terminate.

$a \rightarrow P$ defines a process prefixed with an event using the prefix operator $\rightarrow$. The process will communicate the event a and then behave as process P.

$P; Q$ is the sequential composition operator, which handles the control over Q after P has been successfully terminated.

The process $P \,\square\, Q$ (deterministic choice) and $P \sqcap Q$ (non-deterministic choice) both can behave like P or like Q. The main difference is, in deterministic choice, the environment decides on which process to run.

$a: A \rightarrow P(a)$ denotes a deterministic choice between the events of the set A which may be finite or infinite. This notation allows representing the inputs from and outputs to the channels. The input $c?x : A \rightarrow P(x)$ can accept any input x of type A along channel c, following which it behaves as $P(x)$. Its first event will be any event of the form $c.a$ where $a \in A$. The output $c!v \rightarrow P$ is initially able to perform only the output of v on channel c, then it behaves as P.

The *interrupt* process, $P \vartriangle Q$ defines a process which behaves as P unless it receives an interrupt event, which is the first event of process Q. If this event is received, the control is transferred to Q and process P is discarded.

A *relabeled process* $f(P)$ has a similar control structure to P, however the events are renamed depending on the function f. The *hiding* operator $P \setminus A$ defines a process similar to process P but the environment will not see the events defined in the set A. The hidden events will occur immediately, as the environment is not able to see these events and synchronize with them.

Processes may also be composed in *parallel*. If A is a set of events, then the process $P \underset{A}{||} Q$ behaves as P and Q acting concurrently, with synchronizing on any event in the synchronization set A. Events not in A may be performed by either of the processes independent of the other. If the alphabets of P and Q are A and B respectively, then (provided P and Q cannot communicate using an event outside their alphabets) $P \ {}_A||_B \ Q = P \underset{A \cap B}{||} Q$. The indexed parallel operator $||_{A_P} P$ defines a process which is composed of P processes with a set of respective interfaces A_P. *Interleaving* (i.e. the two components do not interact on any events) is achieved by synchronizing on nothing so $P ||| Q = P \underset{\emptyset}{||} Q$.

The recursion $\mu X \cdot F(X)$ represents a process which behaves like $F(X)$ but with every free occurrence of X in P (recursively) replaced by $\mu X \cdot F(X)$; where the variable X here usually appears freely within $F(X)$.

2.4. Operational Semantics

The operational semantics views CSP processes as transition systems and provides a formal way of generating the transition systems of processes.

Definition (Labeled Transition System): A *labeled transition system (LTS)* is a tuple $(\mathbb{P}, \rightarrow)$ where $\mathbb{P}$ is set of nodes (states) , $\Sigma^{\checkmark,\tau} = \Sigma \cup \{\checkmark, \tau\}$ is set of actions (events) and $\rightarrow$ is a ternary relation such that $\rightarrow \subseteq \mathbb{P} \times \Sigma^{\checkmark,\tau} \times \mathbb{P}$.

A process P, with set of initial states P^0, performs an event from $\Sigma^{\checkmark,\tau}$ and changes its node accordingly. Each node of the LTS represents the state of the process. The LTS can be *finite* or *infinite*, depending on the set of states required to define the process. It is generally assumed that each node is reachable from P^0 (Roscoe 1998).

Firing rules are defined for each CSP operator to systematically generate the state space of processes from process definitions. These firing rules form a logical inference system as a whole. Model checkers use these firing rules to generate the state space of a system (Goldsmith 2005b).

As the CSP processes can be viewed as LTSs, the similarity between LTS and automata raises the question of the relation of CSP with automata. The expressiveness of processes in CSP corresponds to partial automata; in which the transition function is

a partial function (Wolter 2002). Similarly, the expressiveness of Timed CSP is equivalent to a subclass of Timed Automata (Alur and Dill 1994, Ouaknine and Worrel, 2002).

2.5. Behaviors and Denotational Semantics

Denotational semantics of CSP introduces behavioral models for capturing and comparing the behavior of processes. Each model defines the observations to be made on the processes and provides algebraic relations between these observations.

Different behaviors of the system are captured with different models, so they can represent the same system in different levels of detail. What they have in common is that each of them provides an abstract way of representing a process as a set of behaviors it can have in one or more categories.

There are three widely used models in CSP, that represent the most commonly used behaviors, namely Traces ($\mathcal{T}$), Stable Failures ($\mathcal{F}$) and Failures / Divergences ($\mathcal{N}$) models. All three models are based on recording finite observations on processes.

It is also possible to define new behavioral models in CSP which make different observations on processes (Bolton and Lowe 2004, Roscoe, et al. 2007, Roscoe 2008, 2009).

2.5.1. Traces ($\mathcal{T}$) Model

The sequence of all events that a process has communicated by some point in its execution forms the *trace* of the process. A trace of a process may be finite or infinite, but in this model, only finite traces are considered.

$traces(P)$ is the function that defines a process as the set of finite sequences of visible events that it could perform, thus $traces(P) \subseteq \Sigma^*$.

This function must satisfy the following pair of healthiness conditions to be in $\mathcal{T}$ (traces model):

(T1) $\langle\rangle \in traces(P)$

(T2) $tr \,^\wedge\, tr' \in traces(P) \Rightarrow tr \in traces(P)$

The condition (T1) states that empty trace is a trace of process P. (T2) states that the set $traces(P)$ is prefix closed; meaning that any prefix of a trace is also a trace of P. Note that ^ denotes the concatenation of two traces (Roscoe 1998).

The deadlocked process $STOP$ cannot perform any actions, so its trace only contains the empty trace, that is $traces(STOP) = \{ \langle\rangle \}$. On the other hand, the process RUN_Σ is the process that can perform every action in its alphabet Σ, that is $RUN_\Sigma = x{:}\Sigma \rightarrow RUN_\Sigma$. Its traces contain all possible traces in the system: $traces(RUN_\Sigma) = \Sigma^*$.

Traces model is used for general safety specifications of processes.

2.5.2. Stable Failures ($\mathcal{F}$) Model

Stable failures model ($\mathcal{F}$) considers the set of events that a process cannot accept as well as the trace information of a process. $\mathcal{F}$ makes more identifications between processes than $\mathcal{T}$; thus $\mathcal{F}$ is a *finer* model than $\mathcal{T}$.

Definition (Refusal): A *refusal* is set of events that a process fails to accept even if events are available to the process. $refusals(P)$ is the set of P's initial refusals. $refusals(P/tr)$ is the set of P's refusals after a trace of tr, where $refusals(P) \subseteq \mathbb{P}\Sigma$. (Note that $\mathbb{P}$ represents the power set.)

Definition (Stable State): A process is said to be in a *stable state*, if the process has no immediate internal actions (τ). If there are one or more τ's available, these internal actions will be taken and the process gets into some other state; so only stable states have refusal sets.

Definition (Failure): Each possible (tr, X) pair is a *failure* of the process where $(tr, X) \subseteq \Sigma^* \times \mathbb{P}\Sigma$. The function $failures(P) = \{(tr, X) \mid tr \in traces(P) \wedge X \in refusals(P/tr)\}$ forms the failure set of a process P. This function must satisfy both (T1), (T2) (healthiness conditions for $\mathcal{T}$) and these conditions:

(F1) $(tr, X) \in failures(P) \Rightarrow tr \in traces(P)$

(F2) $(tr, X \cup Y) \in failures(P) \Rightarrow (tr, X) \in failures(P)$

(F3) $(tr, Y) \in failures(P) \wedge \forall x \in X \cdot tr ^\wedge \langle x \rangle \notin traces(P) \Rightarrow$
$(tr, X \cup Y) \in failures(P)$

The first condition (F1) provides consistency with the traces of the process, if tr is not a trace of P, then there shouldn't be a failure pair for tr. (F2) condition points out

that every refusal set of possible trace should be subset - closed. The final condition (F3) states that events performed in a particular state can be added to the failures set of the process.

2.5.3. Failures / Divergences ($\mathcal{N}$) Model

While $\mathcal{F}$ can be useful for classifying deterministic and nondeterministic processes, it has some shortcomings when there are processes which include infinite number of internal actions.

Definition (Divergence): A process is said to be *divergent* if infinite, consecutive sequence of internal actions (τ) occur. The set $divergences(P)$ is the set of traces of P (thus, $divergences(P) \subseteq \Sigma^*$) after which the process diverges.

Definition (Deterministic Process): A process P is a *deterministic process* if $divergences(P) = \{\,\}$ and $tr^\wedge\langle a\rangle \in traces(P) \Longrightarrow (tr, \{a\}) \notin failures(P)$, that is; if it does not diverge, it does not accept and refuse the same event.

The sets $traces(P)$ and $failures(P)$ must be extended to provide divergence information:

(1) $traces_\perp(P) = traces(P) \cup divergences(P)$

(2) $failures_\perp(P) = failures(P) \cup \{(tr, X) \mid tr \in divergences(P)\}$

In (1), $traces(P)$ is extended to contain the traces in which the process diverges. (2) states that if a process diverges, then it refuses every event.

In $\mathcal{N}$, a process P can be described with pair $(failures_\perp(P), divergences(P))$ where $failures_\perp \subseteq (\Sigma^{*\checkmark} \times \mathbb{P}\Sigma^{*\checkmark})$ and $D \subseteq \Sigma^{*\checkmark}$. Also, several healthiness conditions are defined for pairs to describe processes:

(F1) $traces_\perp(P) = \{tr \mid (tr, X) \in failures(P)\}$ is non-empty and prefix closed.

(F2) $(tr, X) \in failures(P) \wedge Y \subseteq X \Longrightarrow (tr, Y) \in failures(P)$

(F3) $(tr, X) \in failures(P) \wedge \forall a \in Y \cdot tr^\wedge\langle a\rangle \notin traces_\perp(P) \Longrightarrow$
$(tr, X \cup Y) \in failures(P)$

(F4) $tr^\wedge\langle \checkmark\rangle \in traces_\perp(P) \Longrightarrow (tr, \Sigma) \in failures(P)$

(D1) $tr \in divergences(P) \cap \Sigma^* \wedge\ a \in \Sigma^{*\checkmark} \Longrightarrow tr^\wedge a \in divergences(P)$

(D2) $tr \in divergences(P) \Longrightarrow (tr, X) \in failures(P)$

(D3) $tr\hat{}\langle ✓\rangle \in divergences(P) \Longrightarrow tr \in divergences(P)$

The condition (F2) states that if a process can refuse X, then it can refuse any subset of it. (F3) condition points out that if a process refuses X after tr, then the same state must refuse any events in set Y, the set of events it cannot perform after tr. (F4) states that if a process terminates, then it refuses anything. The divergence condition, (D1) is an extension closure property, stating that if tr is a divergence of P then so is $tr\hat{}a$. Condition (D2) states the process refuses any $X \subseteq \mathbb{P}\Sigma^{*✓}$, because it only performs the internal actions. The last condition, (D3) states the special case of termination event.

Stable processes can be described with same observations in both models. In other words, $\mathcal{N}$ does not provide more observations than $\mathcal{F}$ for stable processes (Roscoe 1998).

2.6. Refinement Relations

Assume that $P \sqcap R$ can be used anywhere instead of R, where P and R are processes. To satisfy this property, P must exhibit all behavior that R exhibits (i.e. every behavior of P should be a behavior of R), therefore it can be said that P *refines* R. That is, if $P \sqcap R = R$ then P refines R, which is symbolized as $R \sqsubseteq P$.

Refinement relation is;

(i) reflexive; for any process P, $P \sqcap P = P$, therefore every process is a refinement of itself

(ii) antisymmetric; for any processes P and R, if $P \sqcap R = R$ and $P \sqcap R = P$ then $P = Q$. This forms the notion of equivalence between processes.

(iii) transitive; for any processes P,R and Q, if $R \sqsubseteq P$ and $P \sqsubseteq Q$, then $R \sqsubseteq Q$.

As the behaviors of processes are defined by models; refinement relations can be expressed in terms of the models. Let $\mathcal{M}[[P]]$ represent the set of all possible behaviors of process P in model $\mathcal{M}$. If process P refines process R in model $\mathcal{M}$, then the set of all behaviors of R should include at least all behaviors of P. That is, if $R \sqsubseteq_{\mathcal{M}} P$, then $\mathcal{M}[[R]] \supseteq \mathcal{M}[[P]]$. In general, *more* behavior a process has, *less* refined it is.

The refinement relation forms a partial ordering between processes. This also means that, there are least refined processes and most refined processes. For example, in $\mathcal{T}$ (traces model), $STOP$ is the most refined process, because for any $P, traces(P) \supseteq traces(STOP)$. The least refined process is RUN_{Σ}, therefore, for any process P, $traces(RUN_{\Sigma}) \supseteq traces(P)$.

It is straightforward to define the refinement relation for $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{N}$ models:

Definition (Traces Refinement): Let P, R be two processes. $R \sqsubseteq_{\mathcal{T}} P$ (P is a trace refinement of R) if and only if $traces(R) \supseteq traces(P)$.

Definition (Failures Refinement): Let P, R be two processes. $R \sqsubseteq_{\mathcal{F}} P$ (P is a failures refinement of R) if and only if $traces(R) \supseteq traces(P)$ and $failures(R) \supseteq failures(P)$. If $R \sqsubseteq_{\mathcal{F}} P$ for processes P and R; then $R \sqsubseteq_{\mathcal{T}} P$ also holds.

Definition (Failures-Divergences Refinement): Let P, R be two processes. $R \sqsubseteq_{\mathcal{N}} P$ (P is a failures - divergences refinement of R) if and only if $failures_{\perp}(R) \supseteq failures_{\perp}(P)$ and $divergences(R) \supseteq divergences(P)$.

Refinement checking involves using another CSP process as the specification process, which identifies the correct behavior of a process. This approach is somewhat similar to the concept of language inclusion, which is defined for finite state automata.

It is also possible to use other formalisms for expressing specifications, such as Computational Tree Logic (CTL) (Emerson and Clarke 1982) and Linear Tree Logic (LTL) (Pnueli 1977). However, it is proved that refinement checking is more expressive than CTL (Leuschel et al. 2001).

In addition to refinement checking, it is possible to use individual properties R and check if a process P satisfies R; which is P ***sat*** R (Hoare 1985). The expressive power of sat based specification and refinement relations is also comparable (Roscoe 2005).

2.7. A Verification Tool for CSP: FDR

Failures - Divergences Refinement (FDR) (Goldsmith 2005a) is a commercial tool, designed to establish verification results for the concurrent and reactive systems

described in CSP. It calculates the state space of the given system and performs an exhaustive search on this space, like the other model checking tools.

Generally, FDR obtains its results by comparing two descriptions: a specification and an implementation; in order to determine if the latter is a refinement of the former. If a check is successful, it means that the implementation is a reasonable candidate for substitution into the role of the specification.

FDR generates LTS of processes according to operational semantics through a process called *compilation*. However, a compilation process that treats all the processes same is inefficient when checking big state spaces. Therefore, FDR uses a two-level approach to calculating operational semantics: The low level is fully general but relatively inefficient, whereas the high level is restricted (specifically, it cannot handle recursion) but much more efficient in terms of space and time.

The most recent version of FDR used in this study is FDR 2.83 which is released on July 23, 2007. FDR can check for refinement in Traces, Stable Failures and Failures – Divergences models and in several non-standard models (Bolton and Lowe 2003, Goldsmith 2005a).

2.7.1. Refinement Checking

The refinement checking mechanism in FDR depends on the theory of fixed point induction (Tarski 1955, Roscoe 1998).

Suppose a check in which whether an implementation process $Impl$ refines the specification process $Spec$ with respect to model $\mathcal{N}$, that is $Impl \sqsupseteq_{\mathcal{N}} Spec$. Let all the states for $Spec$ and $Impl$ be S and I, respectively. Before the refinement check, FDR normalizes specification process $Spec$.

After normalization, a process pair must satisfy these conditions if failures-divergences refinement holds:

(1) $\forall a \cdot (I \xrightarrow{a}) \Rightarrow (S \xrightarrow{a})$

(2) $\forall X \cdot (I\ refuses\ X) \Rightarrow (S\ refuses\ X)$

(3) $I \uparrow \Rightarrow S \uparrow$

The condition (1) states that if an event is possible from implementation state I, then it must be available for specification state S. Condition (2) states that any refusal of

I should be available for S. Since the subset of a refusal is also a refusal, only maximal refusals are checked. Finally (3) states that I should diverge only if S do. (Goldsmith 2005a).

If a pair is correct, then all pairs should be checked, that are reachable from current state. The refinement holds if it holds for every pair in the state space. Hence, refinement checking is only applicable to finite state processes.

2.7.2. Compression Techniques

Similar to other model checkers, FDR also suffers from the state space explosion problem. Some compression techniques are used to decrease the number of states that should be checked. Compression involves converting a state machine into an equivalent but a more efficient state machine, by application of several techniques (Roscoe, et al. 1995, Clarke et al. 1999, Goldsmith 2005a).

These compression techniques are available to all processes because they are applied to the generated LTS by the operational semantics. However the gain provided by these techniques is generally dependent on the structure of the process. Unsuitable compression techniques are known to increase the state space of some processes (Roscoe 1998).

CHAPTER 3

MODELING AND VERIFICATION OF SECURITY PROTOCOLS USING CSP

The purpose of this chapter is to introduce the general concepts and techniques used for modeling and verification of security protocols. In this study, these concepts are heavily used in the modeling of the protocol and verification of the protocol model.

Security protocols (cryptographic protocol) define a sequence of messages between two or more participants. They aim to reach a goal (e.g. authentication between principals, secrecy of a message, etc…) using cryptographic mechanisms such as encryption, decryption and hashing; even if the underlying medium is insecure.

3.1. Stream Authentication

Some applications require a stream of data be transported between agents on a network, often with smaller packets, where the data in each packet is related with the previous and the next packets. This is often the case in real-time stock exchange, audio / video transmission protocols or sensor networks where the data is generated in real time and needs to be transported in a timely manner (Gennaro and Rohatgi 1997, Anderson, et al. 1998, Canetti, et al. 1999). Thus, these applications have different requirements, such as reliability, group management and locus of control (Obraczka 1998).

In addition to these issues, authenticating and signing the stream is fairly different from standard authentication schemes such as Needham – Schroeder Protocol (Needham and Schroeder, 1978) or Yahalom Protocol (Burrows, et al. 1990). These schemes provide authentication between one sender and one responder; which agree on a shared secret key. However, streaming protocols may use many responders while sender is single. So, these schemes are not suitable for streaming applications where the sender must authenticate it to multiple receivers.

Stream authentication protocols are generally verified using theorem proving techniques, due to their infinite state space. However, Timed Efficient Stream Loss-

tolerant Authentication Protocol (TESLA) (Perrig, et al. 2000) has been modeled using CSP and verified using model checking (Broadfoot and Lowe 2002), which used data independence techniques (Lazic 1998, Roscoe and Broadfoot 1999, Roscoe, et al 2000).

3.2. The Generic CSP Protocol Model

Each participant is modeled as CSP processes and a network between these processes is constructed. Security properties of the model are then analyzed with respect to the specifications of the protocol using refinement relations (Ryan et al. 2001).

CSP model of a protocol represents a finite subset of the real protocol, which supposedly includes infinite number of states, to be analyzed by model checking. The network is modeled as a CSP process which is the parallel composition of honest agents, other protocol participants and attacker. The network process provides an abstraction of the real network. Communication between the processes is provided by CSP channels which are labeled with sender and receiver fields. Honest agents use these channels to receive and send the protocol messages. The structure of a generic CSP network model is given in Figure 3.1.

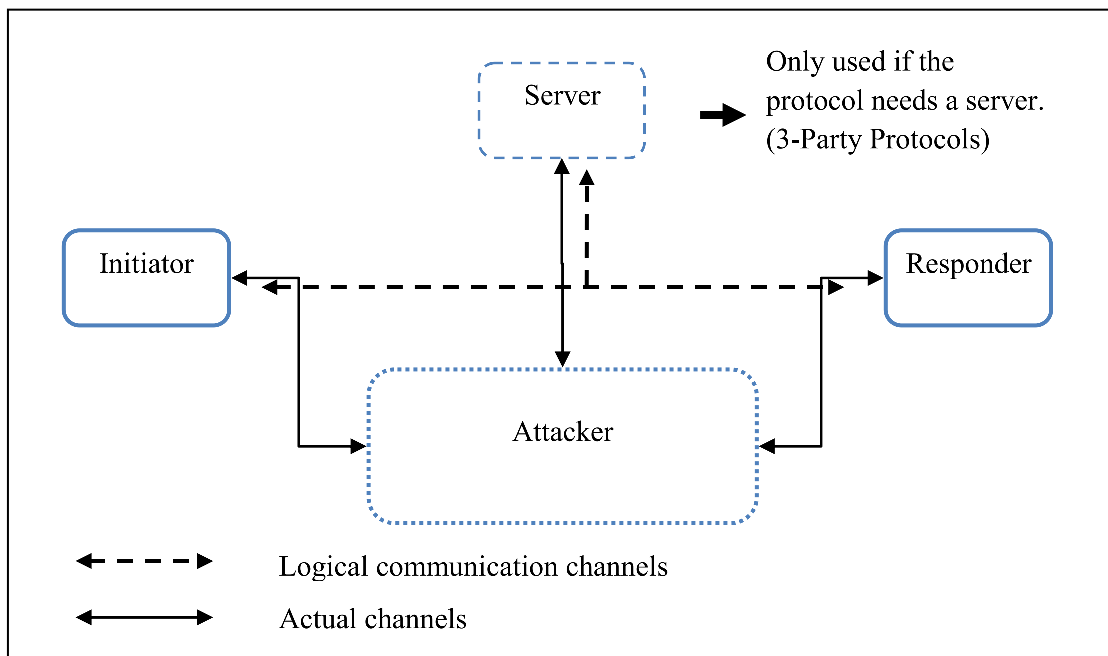

Figure 3.1. Generic CSP Model Used for Verification of Security Protocols

This model can also be modified to satisfy different requirements. For example, more processes can be added to represent agents with different roles or agent processes can contain two or more internal processes (Dilloway and Lowe 2007).

3.2.1. Honest Agents

Honest agents are trustworthy processes that aim to reach the protocol goal, despite the interventions of the un-trusted network. An agent can either act as an *initiator* or *responder* of the protocol and might be parameterized by its identity.

When acting as an initiator, initiating agent chooses an agent to communicate with and then communicates messages of the protocol in turn. The responding process communicates the protocol messages as defined in the protocol description. When the run is complete, agents enter a state in which they are in a session with each other.

In addition to the protocol messages, agents may also engage in signal events. These events represent the state of protocol run and they are sent using different channels such that attacker is not able to interfere with them. Hence, a general view of protocol run can be obtained and this information might be used to build specification processes (Shaikh et al. 2009).

3.2.2. Intruder Model

The network is responsible for the transmission of messages and cannot be trusted, since the messages can be lost, reordered or changed during transmission. Also; the network may contain some malicious parties that may acquire sensitive information and even forge their own messages.

In this model, the intruder is modeled as the network. Capabilities of the intruder are:

- *Deliver Messages:*

  This is basically transmitting the message to intended recipient without any alteration.
- *Overhear Messages:*

Intruder may access the contents of transmitted messages. On overhearing, it may add overheard message to its knowledge base. The message will still be delivered to intended recipient, without alteration.

- *Block (Intercept) Messages:*

  The message will not be delivered to recipient in this case.

- *Forge (Fake) Messages:*

  Intruder may create messages by applying some operations (*deductions*) to messages in its knowledge base. Also; it may send previous messages, which is known as *replay* or *redirection*.

This view of network is known as Dolev - Yao Model (Dolev and Yao 1981) in which a strong intruder that forms the untrusted medium. In this model, the intruder can even perform cryptographic attacks on encrypted messages.

However the protocol must be checked for correctness, not underlying cryptographic operations. Hence the capabilities of intruder are limited by *perfect – cryptology* assumptions.

First, the contents of encrypted messages cannot be viewed, if the process cannot supply the proper decryption key. Additionally, keys cannot be deduced from encrypted messages by guessing or applying cryptanalysis.

This view of network is known as *Dolev - Yao Model with perfect-cryptology*. If the protocol is checked to be correct despite the presence of a strong intruder; it should also be correct for networks with weaker intruders (Roscoe 1995a).

On the other hand; using weaker channels than Dolev - Yao is also possible. In the case of empirical channels (i.e. Channels on which the data is transferred using sight, hearing, etc…), when sight is used as a channel; the channel cannot be spoofed and is authentic, but confidentiality cannot be assured (Nguyen and Roscoe 2008).

3.2.3. Simplifications on Protocol Model

The complexity of protocols makes analysis much more difficult. When analyzing a complex protocol using model checking, state space explosion problem occurs.

The idea is to apply as many safe simplifying transformations as possible, trying to avoid introducing new attacks, and then to analyze the simplified protocol. If the

simplified protocol is secure, then so is the original; if the attack discloses an attack up on the simplified protocol, then it must be considered that whether there is a corresponding attack on the original protocol. If there is such an attack, then clearly the original protocol is flawed; otherwise, it means that the whole scheme has been oversimplified. (Hui and Lowe 1999, 2001).

Examples of such transformations include removing encryptions, hash functions and some atomic or hashed fields, renaming atomic fields and removing fields from the bodies of encryptions (Hui and Lowe 1999, 2001).

3.3. Data Independence

A process can be parameterized if its behavior is dependent on parameters $p_1, p_2, \dots, p_n$ with respective types $T_1, T_2, \dots, T_N$. Each different value bound to parameters will build a different instance of the process. Parameterized processes need to be instantiated using values of appropriate type, in order to be verified using a model checker. Hence, the result of verification is actually meaningful for one instance of system, not for all values of parameterized type.

Definition (Parameterized Verification Problem): The problem of verifying whether a parameterized system satisfies correctness conditions for all parameter values is known as *Parameterized Verification Problem* (Lazic 1998).

Although the Parameterized Verification Problem is undecidable (Krzysztof and Kozen, 1986) there are some subclasses of problems, on which this problem becomes decidable. The data independence techniques focus on identifying such processes.

Definition (Data Independence): A parameterized system P is *data-independent* with respect to a data type T if and only if the operations performed in the system are restricted to:

(i) Input,
(ii) Store and copy within its variables (i.e. polymorphic operations such as tupling),
(iii) Performing equality tests between them (but not ordering),
(iv) Output.

The system must not perform operations on T which can constrain its type, such as computing the size of T, apply ordering operations between members of T, using constants of type T, etc…. Also, the control flow of program must not depend on the parameterized values of type T (Lazic 1998).

Definition (Weak Data Independence): A parameterized system P is *weakly data-independent* if either;

(i) Values from T can affect the control flow of P by equality tests, or,

(ii) It has formal operations from any type to T.

The idea is to verify a data-independent program with respect to type T using the minimum sized set of T; and the result of verification would hold for all finite or infinite sized type T. This minimum size is actually a threshold value; after which the program behavior won't change.

Let $Spec(T)$ and $Impl(T)$ be data independent processes with respect to type T; representing respectively specification and implementation processes. It is needed to determine the threshold values to check for refinement between $Spec(T)$ and $Impl(T)$ and determinism check of process $Impl(T)$.

Definition ($\boldsymbol{NoEqT_T}$ Condition): A data independent process P satisfies $NoEqT_T$ condition if it contains no explicit or implicit equality tests between the members of type T.

If $Spec$ process only knows that an arbitrary member of T is communicated but not interested in which member is; a set of size 1 can be used. Otherwise, if the $Spec$ process is interested in each value is output in right place, a set of size 2 can be used.

For the general case; it is assumed that the data independent processes may have equality tests between members of type T and control flow of process may be affected with these tests. Hence, the $NoEqT_T$ condition might not be satisfied by these processes.

If $Spec(T)$ and $Impl(T)$ are data independent processes with respect to type T and they do not satisfy $NoEqT_T$ condition; thresholds can be computed by determining some constants. These constants need to have a finite value to calculate a usable threshold value; else the process does not have a finite threshold. (Lazic 1998).

3.4. Specification of Security Protocols

This section is devoted to the construction of specification processes for security protocols. There are different methods for specifying secrecy, authentication and non repudiation properties of protocols, however only authentication properties are introduced (Roscoe 1996, 1998, Schneider 1996, 1997, Lowe 1997a).

3.4.1. Using $signal$ Events

It is necessary to capture the state of honest agents during the protocol run in order to build the specifications of the system. Signal events are used to determine this information. Signal events use channels that are not accessible by the intruder. When an agent reaches a certain stage of execution in a run, these events are performed. Then the system's behavior can be captured by hiding other events other than these signal events (Shaikh, et al. 2009).

The channel $signal$ is used to capture the states of execution of agents. An event in the form of $signal.Running.role.A.B.d_1.d_2$ represents that agent A, thinks it is running the protocol as $role$ with B, agreeing on data items d_1 and d_2. This event is performed before the last message that A sends.

An event in the form of $signal.Commit.role.A.B.d_1.d_2$ represents that agent A thinks that it has succeeded in protocol run where it performed as $role$ with B, agreeing on data items d_1 and d_2. This event is performed after A's last event in the run.

3.4.2. Authentication

Authentication is concerned with the verification of an entity's claimed identity. An authentication protocol provides an agent A with a mechanism to ensure that the other party B has also been involved in the protocol run. However, there is no consensus on the exact meaning of authentication. Therefore, four possible specific meanings of the term are introduced and evaluated (Lowe 1997a).

The first definition is the strongest definition; *agreement*, where the protocol agents are required to agree on a mutual set of data items. Also a protocol run of initiator must be matched by a unique run on the responding side of the protocol.

A weaker definition is *non-injective agreement*, where the agents are required to agree on a mutual set of data items, as in agreement, but a run of the protocol might not be matched by a unique run on responding side. A protocol which satisfies agreement specifications also satisfies non-injective agreement specifications.

Weak agreement specifications provide weaker specifications than non-injective agreement. This definition of authentication allows agent B to assume any protocol role in its run. Therefore, agent A does not have a guarantee about the role of responding agents or values of data items. Additionally, two agents do not need to agree on a mutual set of data items.

The weakest definition is *aliveness.* In this definition, a run of agent B may not be recent. In other words, this specification allows checking that agent B has completed the protocol run before agent A. Also agent B may not believe that it was running the protocol with agent A.

CHAPTER 4

PROPOSED MODEL FOR THE PROTOCOL

In this chapter, the EMSS protocol is introduced and evaluated under CSP for verification. First, infinite state CSP model of the protocol is built, which is then reduced into a finite state model using some observations and assumptions while retaining the properties of the protocol. This model is represented as a CSP process that describes the behavior of the protocol.

4.1. The Efficient Multi-chained Stream Signature (EMSS) Protocol

The Efficient Multi-chained Stream Signature (EMSS) protocol (Perrig, et al. 2000) aims to provide the authentication of sender to receivers, which is based on signing a small number of special messages in a data stream. Each packet is linked to a signed message via multiple hash chains.

Hash chains are formed by appending the hash of each message to a number of subsequent messages.

The main features of this scheme are; the amortization of the cost of signature operation over multiple packets, resistance to packet loss, low message overhead and non repudiation of the sender to the transmitted data (Perrig, et al. 2000).

The hash chains are formed by appending the hashes of a number of previous messages to current message P_i. Periodically, a signature message P_{sign} is sent, which contains the hashes of previous messages in a signature. Receiver then decrypts the signature message and obtains the signed hashes of previous messages. The chained hashing relation between the messages is illustrated in Figure 4.1.

There should be a path from a message to the signed message to authenticate the message. To authenticate message P_i, then the existence of messages P_{i+1} or P_{i+2} is sufficient. If both of these messages are lost, then there's no path to P_i from the signature message and all $P_k \ where \ 0 \leq k \leq i$ cannot be verified.

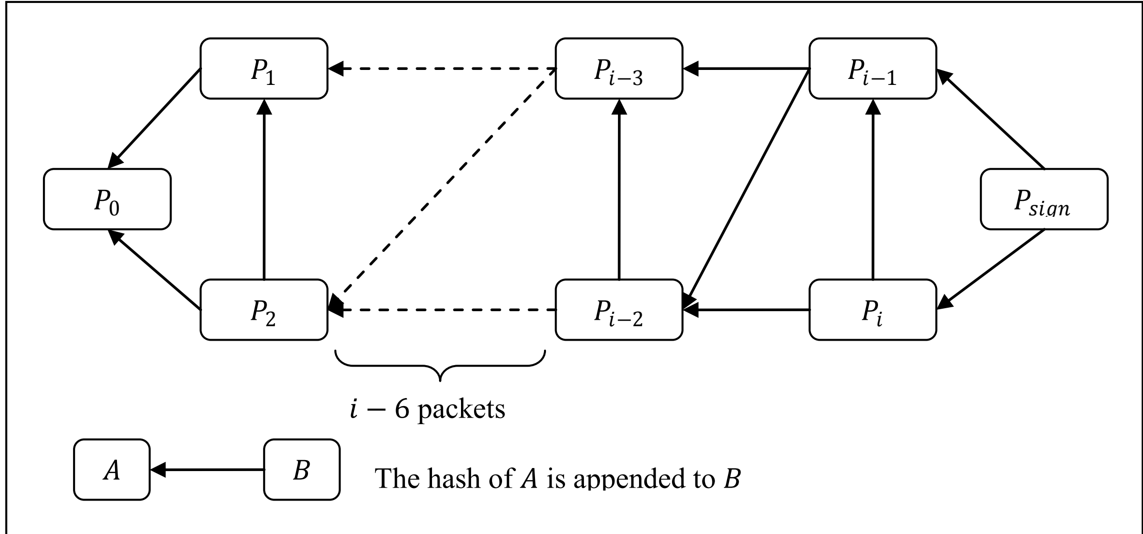

Figure 4.1. The Hash Chains Between the Protocol Messages in EMSS

The digital signature message can be sent periodically for every n messages, which would be different for each application of protocol. Clearly, for authenticity verification, the receiver must buffer messages until a signature message arrives. Several use cases are defined for setting these parameters to use the protocol for different purposes (Perrig, et al. 2000).

EMSS can be represented in Security Protocol Notation as:

1. $A \to B : P_0,\ P_0 = \mathrm{d}_0$
2. $A \to B : P_1,\ P_1 = d_1, h(P_0)$

i. $A \to B : P_i,\ P_i = d_i, h(P_{i-1}), h(P_{i-2})$ where $2 \leq i < n$

… …

n. $A \to B : P_n,\ P_n = \{h(P_{n-1}), h(P_{n-2})\}_{sk(A)}$

where, d is the data and h is the hash function. Since the protocol is not defined in security protocol notation (Perrig, et al. 2000), the protocol definition that is used in this study is the same as the one in (ter Beek, et al. 2006).

Only d_0 is sent to the receiver as the first protocol message P_0. In the second message, d_1 is sent to the receiver along with the hash of d_0. Next message consists of d_2 and hashes of P_1 and P_0. Thus, P_0 may be still authenticated even if P_1 is lost or unauthenticated. The last message consists of P_{n-1} and P_{n-2}, encrypted with $sk(s)$ - the private key of the sender.

As opposed to non-stream protocols, which have a predefined number of protocol messages in a run, the maximum message number in a run of EMSS protocol depends on the application (Perrig, et al. 2000).

EMSS protocol does not make any secrecy claims, as the data items are sent to the receiver in the clear. However, the receiver is able to check for the authenticity of incoming messages when a signature message arrives. Thus, EMSS claims to provide *one-way authentication*, only authenticating the sender to receivers. The sender agent is not able to authenticate the receiver agents. In this study, one-way authentication properties of EMSS protocol are considered.

4.2. Modeling Assumptions

It is assumed that the agents perform only the operations defined in the protocol description. Also, the agents are assumed to know their own secret keys, the public keys and the identities of other agents. One further assumption is that the intruder has the capabilities of a Dolev - Yao Intruder (Dolev and Yao 1981).

Additionally, a perfect cryptosystem is assumed in order to focus the analysis on the protocol. For hashing, it is assumed that agents cannot extract the message contents from hashed messages. Also, hashes of different messages are always different. Hence, a perfect hashing scheme is assumed where the hashed message gives no clue about the clear message and furthermore, no hash collisions occur.

The protocol definition does not include any communication from the receiver to the sender. So it is assumed that the sender knows the identities of receivers with which it will communicate with. In reality, this is likely to be a subset of all agent identities.

EMSS protocol can be configured to have more than two hash values in a protocol message (Perrig, et al. 2000). In this study, it is assumed that one message has the hashes of two other messages.

4.3. Symbolic Data Types and Operations

Symbolic data types model the actual data structures and variables within the protocol. This abstraction provides a degree of freedom from the constraints on actual data structures but allows including them in the model.

Cryptographic operations are also represented symbolically for symbolic data types, thus, the operation is not performed in reality; but relevant symbolic data are modified appropriately. This allows the model to ignore the implementation specific details of cryptosystem, as stated in the modeling assumptions.

4.3.1. Symbolic Data Types

Protocol messages and their contents need to be represented in a structured way. Messages and their subcomponents are constructed from atomic data items by concatenation. This means that a communication of a compound message is equivalent to the communication of all of its atomic subcomponents.

Hence, a data type $fact$ is defined in CSP_M, in order to represent atomic data items such as agent identities, keys, encrypted and hashed messages and compound data items that are built using atomic data items:

$$
\begin{aligned}
datatype\ fact\ =\ & Agent.AGENT \mid Sq.Seq(fact) \mid \\
& pk.AGENT \mid sk.AGENT \mid key.(AGENT, AGENT) \mid \\
& Pk.(fact, fact) \mid Encrypt.(fact, fact) \mid Hash.(fact)
\end{aligned}
$$

The $AGENT$ set contains the agent labels. An agent A is identified by $Agent.A$ in protocol messages, where $A \in AGENT$.

Sequenced messages can be represented by $Sq.\langle fact_0, fact_1, \ldots, fact_n \rangle$, which represents a sequence of facts; $fact_0$ to $fact_n$. The public and secret keys of agents are represented respectively by $pk.AGENT$ and $sk.AGENT$. Hence, a key pair for an agent A becomes $pk.A$ and $sk.A$; which are the *dual* of each other. Similarly, $key.(AGENT, AGENT)$ symbolizes a symmetric key between two agents.

The data type $fact$ is defined recursively to allow the use of nested encryption, nested hashing and sequencing inside an encryption.

The set of items that can be constructed from $fact$ has unnecessary and invalid combinations of facts, most of which don't represent meaningful protocol messages. For example, $Encrypt.(key.(A, B), \ldots)$ and $Encrypt.(key.(B, A), \ldots)$ represent the same message, because $key.(A, B)$ and $key.(B, A)$ represent the same key. In a similar sense, illegal protocol messages can be constructed using this set.

These definitions are restricted in order to reduce the state space; by defining a finite set of protocol messages. This approach helps reducing the need of recording every message separately, providing a reduction in the number of facts to be tracked.

4.3.2. Cryptographic Operations

In the analysis, the main focus of interest is finding attacks which are mounted on the behavior of protocol participants, not on the cryptosystems used in the protocol. Furthermore, CSP is not a suitable formalism to represent and verify cryptosystems.

Therefore encryption, decryption and hashing operations are represented as symbolic operations. The operation is not performed in the real sense, only the data items are tagged (or untagged) to symbolize that the operation is performed.

In symbolic encryption, $Pk.(fact, fact)$ represents a public key encrypted message and $Encrypt.(fact, fact)$ represents a symmetrically encrypted message. Using the values within an encrypted data block means the decryption of the message. If the receiving agent knows $key.(A, B)$, it can receive (synchronize on) the message and use its contents. However, the intruder should have to decode the message if and only if it has the right key. Decryption is provided in a deduction in this case; the intruder does not have access to the contents unless it has the corresponding key.

Hashing mechanism is similar to this approach. However, intruder does not have any deductions to retrieve the contents of the hashed messages.

4.4. Modeling the Hash Chain Mechanism using Fixed Hash Sizes

This section discusses problems encountered during the modeling of EMSS protocol and proposes a method to overcome these problems. First, the straightforward approach and its problems are explained. Then, a fixed size hashed message approach is defined for hash chain modeling.

The straightforward approach expresses the hash values by tagging the data values with $Hash.(\,)$. This does not pose a problem for non-stream protocols, which do not use nested hashing. However, as in the EMSS protocol, the length of the representation of the hash value becomes very large when nested hashing is used. Using this straightforward method, i^{th} EMSS protocol message is represented as:

$send.Alice.Bob.Sq.\langle d_i,$

$$Hash.(Sq.\langle d_{i-1}, Hash.(Sq.\langle d_{i-2}\rangle \dots)\rangle),$$
$$Hash.(Sq.\langle d_{i-2}, Hash.(Sq.\langle d_{i-3}\rangle \dots)\rangle)$$

$\rangle$

where $Alice, Bob \in AGENT$.

Another issue is the types of $send$ and $receive$ channels. In CSP; it is possible to define channels with arbitrary type; however for model-checking purposes; the types of channels should be defined explicitly. Channels which accept messages with arbitrary number of nested hashes are a problem while expressing the protocol for model-checking tools.

Because of these reasons, hash chaining in the EMSS protocol using a tagged representation cannot be modeled for model checking purposes.

4.4.1. Using Fixed Hash Sizes

In this approach, the size of a hashed message should be fixed, whatever the message length is. So, i^{th} protocol message, sent from $Alice$ to Bob is represented by:

$$send.Alice.Bob.Sq.\langle d_i, h_{i-1}, h_{i-2}\rangle$$

where $h_{i-1}$and h_{i-2} represent the hash values of previous two messages.

Representing the protocol messages in such a manner requires a relationship, a mapping between the hash values and the protocol messages to be hashed. The receiver process needs to check the correctness of the hash values by comparing received hashes with computed hashes, thus such a mapping becomes necessary.

Before the definition of mapping data, protocol messages and hashes should be represented using sets, whose elements are symbolic representations of actual data structures. Elements of the $AllData$ set represent all possible symbolic data values in the protocol:

$$AllData = \{d_i \mid i \in \{0..m\}\}$$

where m represents the upper bound on the number of data values. The $AllHashes$ set represents all symbolic hash values usable on the system, with n being the maximum number of symbolic hashes:

$$AllHashes = \{h_i \mid i \in \{0..n\}\}$$

Finally, the set of all possible protocol messages is represented by PM set, which use $AllData$ and $AllHashes$ sets as building blocks. The structure of protocol messages vary, the 1st and 2nd protocol messages have zero and one hash values appended, but other messages have two hash values appended. Hence, PM_1 and PM_2 sets represent all possible 1st and 2nd messages, while PM_C set defines all remaining messages. The nth message (i.e. the signature message) is excluded from this set as its hash is not used in the protocol.

$$PM = PM_1 \cup PM_2 \cup PM_C$$

$$PM_1 = \{ \langle d \rangle \mid d \in AllData\}$$

$$PM_2 = \{ \langle d, h \rangle \mid d \in AllData, h \in AllHashes\}$$

$$PM_C = \{ \langle d, h_1, h_2 \rangle \mid d \in AllData, h_1, h_2 \in AllHashes\}$$

PM is the set of all possible protocol messages, although many combinations of the messages are invalid. However, it is important because its union with PM_N set used as the type of $send$ and $receive$ channels where $PM_N = \{Pk.(sk.a, \langle h_1, h_2 \rangle) \mid a \in Agent,\ h_1, h_2 \in AllHashes\}$ is the set of all possible signature messages.

4.4.2. The Construction of $DataHash$ Set

The PM set contains all valid and invalid protocol messages, however a mapping between $AllHashes$ and PM set is still required to define the $hash$ relation.

This relationship is implemented by a mapping set $DataHash$, which contains all protocol messages paired with a distinct hash value. So, $DataHash$ set is composed of pairs (h, m) where $(h, m) \subseteq AllHashes \times PM$. A message can only use hash values which have already been assigned to one of previous messages, which is illustrated in Figure 4.2.

Similar to the definition of the PM set, the $DataHash$ set can be defined as the union of $DataHash_i$ sets, where the subscript i represents the ith protocol message.

For $i = 0$, $DataHash_0$ set contains tuples (h, m) where $(h, m) \subseteq AllHashes \times PM_1$, that is $(h_x, \langle d_x \rangle)$ and $h_x \in AllHashes, x \in \{0 \dots m\}$. The $DataHash_0$ set only enumerates the hash values with protocol messages, defining different hash value for each different data item:

$$DataHash_0 = \{ (h_x, \langle d_x \rangle) \mid x \in \{0 \dots m\} \}$$

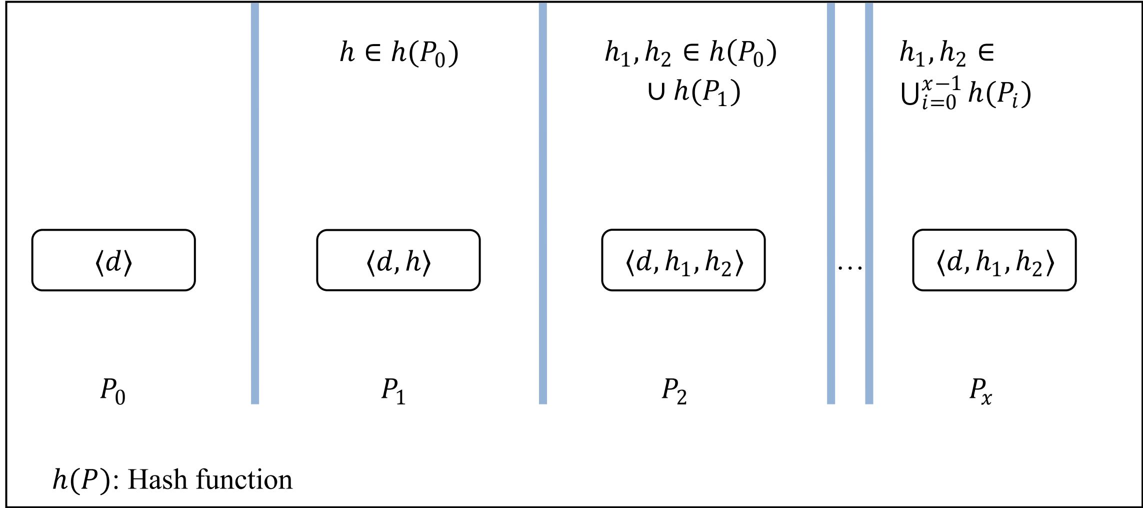

Figure 4.2. Each Message Must Use the Hashes of Preceding Messages

As there are a total of m distinct data values, there are m different hash values corresponding to the hashes of these data values for $DataHash_0$.

For $i = 1$, $DataHash_1$ set contains tuples (h, m) where $(h, m) \subseteq AllHashes \times PM_2$, that is $(h_x, \langle d_y, h_z \rangle)$ and $h_x \in AllHashes, y, z \in \{0 \ldots m\}$. The values of h_z are the hash values defined in the previous set $DataHash_0$. Similarly, this set enumerates the hash values with 2nd protocol messages, but this time, it associates every distinct data and hash value pair with a unique hash value denoted with the subscript x:

$$DataHash_1 = \{ \left(h_x, \langle d_y, h_z \rangle\right) \mid 0 \leq y < m,\ \ 0 \leq z < m \}$$

where $x = |DataHash_1|(y + 1) + z$.

For the general case, where $i \neq 0$ and $i \neq 1$, the respective $DataHash_i$ sets are composed of tuples (h, m) where $(h, m) \subseteq AllHashes \times PM_C$. That is, the structure of each tuple is $(h_x, \langle d_y, h_z, h_t \rangle)$ where $h_x \in AllHashes$ and $\langle d_y, h_z, h_t \rangle \in PM_C$.

The calculation of lower and upper bounds for z and t need more attention. A message at $i \geq 3$ could use hash values defined in sets beginning from $DataHash_0$ to $DataHash_{i-1}$. Thus, the upper bounds for z and t for any message would be the total number of hash values assigned until (including) $i - 1^{th}$ message. Let this number be k, and its value is equal to $\sum_{c=0}^{i-1} |DataHash_c|$.

The lower bounds of z and t are fixed to be 0 for $DataHash_i$, which has several implications. First of all, this definition causes collisions in $DataHash$ set. For example the hash of $\langle d_0, h_0, h_0 \rangle$ is h_6 for $DataHash_3$ and h_{78} for $DataHash_4$, but the relation

that needs to be provided is one-to-one. Another implication is the size of $DataHash$ set is not at its minimum, as there are colliding entries in the set.

For colliding entries, it is safe to assume that $hash$ function returns the first entry in the set. That is, $hash(\langle d_0, h_0, h_0 \rangle)$ returns h_6, not h_{78} or any other value, providing the necessary one-to-one relationship between the hash values and protocol messages.

Another solution for this problem can be the redefinition of $DataHash_i$ set as the union of 3 subsets, each containing different bounds for z and t. In this case, first subset should have the boundaries $0 \leq z < k'$ and $k' \leq t < k$, next one should have $k' \leq z < k$ and $0 \leq t < k$ and the last one $k' \leq z, t < k$; where k' is the total number of hash values assigned until $i - 2^{th}$ message, that is $k' = \sum_{c=0}^{i-2} |DataHash_c|$.

Hence, the boundaries of z and t are assumed as $0 \leq z, t < k$:

$$DataHash_i = \{ (h_x, \langle d_y, h_z, h_t \rangle) \mid 0 \leq y < m, 0 \leq z, t < k \}$$

where $x = k^2 y + k(z + 1) + t$ and $k = \sum_{c=0}^{i-1} |DataHash_c|$.

The $DataHash$ set with a maximum length of n is composed of union of all sets and represents all possible hash values of protocol messages:

$$DataHash = \bigcup_{i=0}^{n} DataHash_i$$

Thus, the $DataHash$ set is formed as the union of all preceding sets and contains all possible and valid combinations of messages and hashes.

4.4.3. The $hash$ function

The $hash$ relation is actually a one-to-one function $hash: PM \rightarrow AllHashes$ which transforms a given protocol message into a hash value, because it is a one-to-one and total relation, with the domain PM and range $AllHashes$. That is, for every protocol message, there is a unique hash value defined.

The $hash(\ ds\ )$ function is defined to return the unique h_i value assigned to a particular ds from $DataHash$ set:

$$hash(ds) = \{h' \mid (h', s) \in DataHash, s = ds\}$$

This hashing scheme obeys modeling assumptions about hashes; processes cannot retrieve the contents of the messages only the hash values. Also, this mechanism constructs the mapping $DataHash$ as one-to-one, so, ensuring all hashes in the system are distinct, for distinct messages.

To illustrate the use of fixed sized hashes, below is an example message exchange between protocol agents. Assume that two distinct data values and hash chain length of 4 is used, that is $AllData = \{d_0, d_1\}$ and $DataHash = \bigcup_{i=0}^{3} DataHash_i$. Let another assumption be that sender sends the data in this order:

$$\langle d_0, d_1, d_1, d_0 \rangle$$

In this case, the contents of $DataHash$ is computed by computing the contents of each set $DataHash_0, \dots, DataHash_3$:

$DataHash_0 = \{ (h_0, \langle d_0 \rangle), (h_1, \langle d_1 \rangle) \}$
$DataHash_1 = \{ (h_2, \langle d_0, h_0 \rangle), (h_3, \langle d_0, h_1 \rangle), (h_4, \langle d_1, h_0 \rangle), (h_5, \langle d_1, h_1 \rangle) \}$
$DataHash_2 = \{ (h_6, \langle d_0, h_0, h_0 \rangle), \dots, (h_{40}, \langle d_1, h_0, h_4 \rangle), \dots, (h_{77}, \langle d_1, h_5, h_5 \rangle) \}$
$DataHash_3 = \{ (h_{78}, \langle d_0, h_0, h_0 \rangle), \dots, (h_{522}, \langle d_0, h_4, h_{40} \rangle), \dots, (h_{12245}, \langle d_1, h_{77}, h_{77} \rangle) \}$

For the first message, only the data item is sent, thus $P_0 = d_0$. For the second message, $P_1 = \langle d_1, hash(P_0) \rangle$, and $hash(P_0) = hash(\langle d_0 \rangle) = h_0$, so $P_1 = \langle d_1, h_0 \rangle$. The third message is $P_2 = \langle d_1, hash(P_0), hash(P_1) \rangle = \langle d_1, h_0, h_4 \rangle$. Finally, following the same pattern, the final message is $P_3 = \langle d_0, hash(P_1), hash(P_2) \rangle = \langle d_0, h_4, h_{40} \rangle$.

Only the hash of P_3 is required for the signature message, which is already present in $DataHash_3$ set. So, $\langle hash(P_2), hash(P_3) \rangle = \langle h_{40}, h_{522} \rangle$ becomes the signature message which is ready to be encrypted with the private key of the sender.

4.4.4. Implementation Considerations

In this section, the CSP_M representation of hash chains are defined using the fixed sized hash values. First of all, the datatype definitions need to be extended to allow representation of hash and data values with subscripts.

First, the $Hash.()$ tag is removed from the datatype definition. A hash value h_i is represented by $h.i$ in CSP_M, where $i \in \{0..m\}$. The $AllHashes$ and $AllData$ sets contain symbolic hash and data values: $AllHashes = \{h.i \mid i \in \{0..n\}\}$ and $AllData =$

$\{d.i \mid i \in \{0..m\}\}$. In the implementation, the items of sets $AllData$ and $AllHashes$ belong to the type $fact$ and the sets of subscripts are symbolized by $HASH$ and $DATA$ sets (Figure 4.3).

$$
\begin{aligned}
datatype\ fact\ &=\ Sq.Seq(fact) \mid pk.AGENT \mid \\
&\quad\ sk.AGENT \mid Pk.(fact, fact) \mid \\
&\quad\ Alice \mid Bob \mid Cameron \mid h.HASH \mid d.DATA \\
AGENT\ &=\ \{Alice, Bob, Cameron\} \\
HASH\ &=\ \{0..n\} \\
DATA\ &=\ \{0..m\}
\end{aligned}
$$

Figure 4.3. Actual Data Types Used in the Protocol Model

The PM set is constructed from $HASH$ and $DATA$ sets, which was defined in Section 4.4.1. The signature message is appended in CSP_M representation in Figure 4.4.

$$
\begin{aligned}
PM_0\ &=\ \{\ \langle d.x \rangle \mid x <- DATA\ \} \\
PM_1\ &=\ \{\ \langle d.x, h.y \rangle \mid x <- DATA, y <- HASH\ \} \\
PM_c\ &=\ \{\ \langle d.x, h.y, h.z \rangle \mid x <- DATA, y\ <- HASH, z <- HASH\ \} \\
PM_n\ &=\ \{\ Pk.(sk.a, \langle h.x, h.y \rangle) \mid a <-AGENT, x <- HASH, y <- HASH\ \} \\
PM\ &=\ Union(\{\ PM_0, PM_1, PM_c, PM_n\ \})
\end{aligned}
$$

Figure 4.4. The PM Set Including the Signature Message

The implementation of $DataHash$ set is more complicated. To assign the subscripts of hash values, $Card_DataHash(i)$ function is defined (Figure 4.5).

$$
\begin{aligned}
&cD = card(DATA)\\
&Card_DataHash(i) = if\ i == 1\ then\\
&\qquad\qquad cD\\
&\qquad else\ if\ i == 2\ then\\
&\qquad\qquad cD * (cD + 1)\\
&\qquad else\ let\\
&\qquad\qquad\qquad di = Card_DataHash(i-1)\\
&\qquad\qquad within\\
&\qquad\qquad\qquad cD * di * di + di
\end{aligned}
$$

Figure 4.5. The Function that Computes the Cardinality of $DataHash$

The $Card_DataHash(i)$ function is a recursive function, which calculates the cardinality of $DataHash_i$ set. The value returned by the $Card_DataHash(i)$ function corresponds to the k value, which is defined in Section 4.4.2 and used in construction of $DataHash_i$ set (Figure 4.6).

$$
\begin{aligned}
&DataHash(1) = \{ (h.x, \langle d.x \rangle) \mid x <- DATA \}\\
&DataHash(2) = \{ (h.(cD(x+1)+y), \langle d.x, h.y \rangle) \mid\\
&\qquad\qquad x < -DATA, y < -DATA \}\\
&DataHash(i) = let\\
&\qquad\qquad cDi = Card_DataHash(i-1)\\
&\qquad within \{\\
&\qquad\qquad (h.(cDi * cDi * \mathrm{y} + \mathrm{cDi} * (\mathrm{z}+1) + t), \langle d.x, h.y, h.z \rangle) \mid\\
&\qquad\qquad x <- DATA,\ y <- \{0..cDi-1\},\ z <- \{0..cDi-1\}\\
&\qquad \}\\
&DataHash = Union(\{DataHash(1), DataHash(2), DataHash(3)\})
\end{aligned}
$$

Figure 4.6. The $DataHash$ Set of Size 3

4.5. Infinite State CSP Model of the EMSS Protocol

Due to the fixed sized hash values, a CSP model of the protocol could be constructed. In this phase, the state space generated from the model is not considered, only the model is built.

As a first step, the $send$ and $receive$ channels are defined. These channels are used by the CSP processes of honest agents. Then the $Intruder$ process is built, which forms an operational attacker based on the Dolev-Yao Intruder Model.

Finally, $send$ and $receive$ channels are connected to $Intruder_Basic$ process, which represents a reliable transmission medium, transmitting messages without any modification.

The EMSS protocol defines two honest agents, one of which is the $Sender$, responsible for sending a stream of data through the network and the other one is the $Receiver$, responsible for collecting the data, checking the received data and authenticating them. $Send_0(id)$ and $Recv_0(id)$ processes represent honest agents whose identities are defined by the parameter id.

4.5.1. Sender Agent

The sender agent is responsible for calculating the hashes of previous messages for preparing and transmitting the protocol messages to the receiver. The sender process is composed of three sub processes: First process is responsible for sending 1st and 2nd protocol messages, which have a different format than i^{th} message. Second process sends i^{th} message and the last process sends the signature message.

The first part chooses a receiver to communicate with, receives the data from $DataStream$ process and forms and transmits protocol messages using the $send$ channel.

After sending initial protocol messages, the identity of the receiver agent is passed to the following process along with the hashes of the previous messages. Note that, $P_0 = d_0$ and $P_1 = \langle d_1, hash(d_0) \rangle$:

$$Send_0(id) = \sqcap\ b{:}\ Agent\ \bullet$$
$$getData.id?d_0 \rightarrow send.id.b.d_0 \rightarrow$$

$$getData.id?\,d_1 \rightarrow$$
$$send.id.b.Sq.\langle d_1, hash(d_0)\rangle \rightarrow$$
$$Send_i\big(id, b, hash(P_0), hash(P_1)\big)$$

The second part requests for another data item, hashes the old data items and sends the ith protocol message. Alternatively it may choose to send the signature message, which is chosen nondeterministically:

$$Send_i(id, b, h_1, h_2) \;=\; Send_n(id, b, h_1, h_2) \;\sqcap\; (getData.id?\,d_i \rightarrow$$
$$send.id.b.Sq.\langle d_i, h_1, h_2\rangle \rightarrow Send_{i+1}\big(id, b, h_2, hash(P_i)\big)$$

The final part of the sender agent involves sending the signature message. A new run of the protocol can begin after sending the signature message:

$$Send_n(id, b, h_1, h_2) = send.id.b.Pk.(sk.id, Sq.\langle h_1, h_2\rangle) \rightarrow Send_0(id)$$

Note that in a fresh protocol run, old hash and data values are assumed to be forgotten by the sender. Hence the processed data and the hash items are not sent as parameters to the new protocol run.

4.5.2. Receiver Agent

Similar to the sender, the receiver is modeled using three sub processes, each having similar roles with their $Sender$ counter parts.

It is important to note that the set R has the structure of (i, m) where $i \in \mathbb{N}$ is the sequence number of arrived message and $m \in PM$ is the message. This structure is required by the $Check$ process, because, it needs to know the sequence number of a stored message.

However, unlike the sender process, the receiver should also check the validity of the received protocol messages after the reception of a signature message. The messages are just buffered in the set $Recvd$ until a signature message arrives:

$$Recv_0(id) \;= \Box\, a{:}\,Agent, dr_0{:}\,AllData \;\bullet$$
$$recv.a.id.dr_0 \rightarrow$$
$$\Box\, hr_0{:}\,AllHashes, dr_1{:}\,AllData \;\bullet$$
$$recv.a.id.Sq.\langle dr_1, hr_0\rangle \rightarrow$$
$$Resp_i(id, a, \{(0, \langle dr_0\rangle), (1, \langle dr_1, hr_0\rangle)\})$$

The second process is responsible for collecting any i^{th} message sent by the sender. The sender might choose to send the signature message after 2nd message, so this part should be prepared to receive a signature message.

$$
\begin{aligned}
&Recv_i(id, a, Recvd) \;=\; Recv_n(id, a, Recvd) \,\square\, (\\
&\qquad \square\, hr_1, hr_2 \,:\, AllHashes, d_r \,:\, AllData \bullet \\
&\qquad\qquad recv.a.id.Sq.\langle d_r, hr_1, hr_2 \rangle \rightarrow \\
&\qquad\qquad Recv_{i+1}(id, a, Recvd \cup \{(i, \langle d_r, hr_1, hr_2 \rangle)\})
\end{aligned}
$$

The final process is responsible for the reception of the signature message.

$$
\begin{aligned}
&Recv_n(id, a, Recvd) = \;\square\, hr_0, hr_1 \,:\, AllHashes \;\bullet \\
&\qquad recv.a.id.Pk.(sk.a, Sq.\langle hr_0, hr_1 \rangle) \rightarrow \\
&\qquad Check(Recvd, \{(n, \langle hr_0, hr_1 \rangle)\}, n-1)
\end{aligned}
$$

4.5.3. Check Process

EMSS protocol does not explicitly specify a method for checking the authenticity of received messages, allowing assumptions about the behavior of an agent. For this reason, the process of checking the incoming messages is as explained below.

After the arrival of the signature message, the buffer should be checked for (possibly) valid messages. The buffer of received packets (i.e. $Recvd$ set in $Recv$ process, R in check process), set of validated messages (i.e. the set V) that initially has the hashes in the signature message and the total number of received packets n are passed as parameters to C$heck$ process. This process uses the algorithm defined in Figure 4.7 to determine the validity of each message.

```
input:  received messages R = {P_0 ..., P_{n-1}}, valid messages V = {P_n}
output: V containing all valid messages
        n ← |R|
        for i ← n to 0 do
                if ((P_{i+1} ∈ V) and hash(P_i) = hp_1(P_{i+1} )) or
                   ((P_{i+2} ∈ V) and hash(P_i) = hp_2(P_{i+2} )) then
                        V ← V ∪ {P_i}
                end
                R ← R − {P_i}
        end
output V
```

Figure 4.7. The Algorithm of $Check$ Process

Basically, this algorithm begins from the $(n-1)^{th}$ message and for each message P_i in the received set R, and checks whether any of the messages P_{i+1} and P_{i+2} are in validated set. A message cannot be verified, if there is not a link to at least one validated message or the hash values in the validated messages are not equal to current message's hash, and discarded from set R. The verified messages are stored in the validated set V, which contains all verifiable messages when the algorithm terminates.

The helper functions $hp_1(P_i)$ and $hp_2(P_i)$ return first or second hash value in P_i, respectively. That is, $hp_1(P_i) = h_{i-1}$ and $hp_2(P_i) = h_{i-2}$ if $P_i = \langle d_i, h_{i-1}, h_{i-2} \rangle$. The function $data(P_i)$ returns the data part of the message P_i.

This algorithm is represented as a recursive process, with slight differences. If a message is valid, it is sent via $putData$ channel as well as addition to the verified set V and removal from the message buffer set R. Additionally, the process needs to be able to receive further messages from $Sender$, after finishing checking.

$$
\begin{aligned}
& Check(R, V, i) = \\
& \quad if\ i < 0\ then \\
& \quad\quad Recv_0(id) \\
& \quad else \\
& \quad\quad if\ \big((P_{i+1} \in V)\ and\ hash(P_i) = hp_1(P_{i+1}\ )\big) \\
& \quad\quad\quad or\ \big((P_{i+2} \in V) and\ hash(P_i) = hp_2(P_{i+2}\ )\big)\ then \\
& \quad\quad\quad\quad putData.data(P_i) \rightarrow\ Check(R - P_i, V \cup P_i, i - 1)
\end{aligned}
$$

$else$

$$Check(R - P_i, V, i - 1)$$

Note that, if more than two consecutive messages cannot be verified, then the second if condition will never be satisfied by subsequent recursions, which matches with the definition of the EMSS protocol (Section 4.1).

4.5.4. Intruder and the Network

In this study, two intruder models are used. The first model, which is known as the basic model, redirects data from $send$ channel to $recv$ channel and does nothing else. The second model forms a complete intruder. These two models are explained in the following section.

4.5.4.1. Basic Intruder Model

The basic intruder model only acts as a medium that delivers messages without any alteration or recording, which is modeled as $Intruder_B$ process. This process is important in the validation of the agent processes.

The basic intruder can be defined as:

$$Intruder_B = \Box\, a, b : Agent, m: PM \bullet$$
$$send.a.b.m \rightarrow recv.a.b.m \rightarrow Intruder_B$$

4.5.4.2. Full Intruder Model

To detect any attacks, the full intruder model is used in the verification of the protocol model, which has the capabilities of Dolev – Yao Intruder model.

Intruder initially knows all public facts and the private facts about an agent that it can impersonate. Two additional channels are used, $hear$ and say, to represent communications of the intruder. The type of $hear$ and say channels is PM, which is the set of all protocol messages:

$channel\ hear, say: PM$

Intruder is allowed to overhear messages and add them to its knowledge base and send messages from its knowledge base.

$$Intruder(IK) =$$
$$hear?m \rightarrow Intruder\left(\overline{IK \cup \{m\}}\right)$$
$$\sqcap\ say!m : IK \rightarrow Intruder(IK)$$

The operation $\overline{IK \cup \{m\}}$, is a closure operation under deduction relation:

$$\bar{S} = \begin{cases} S & S^{'} - S = \emptyset \\ \overline{S \cup S^{'}} & otherwise \end{cases}$$

where $S^{'} = \{f \mid (X, f) \in AllDeductions,\ S - X = \emptyset\}$. That is, $S^{'}$ is the set of facts that can be inferred from the power set of elements of S. The set $AllDeductions$ are set of all deductions that can be done by using facts.

The knowledge base of the intruder is not stored as a set; rather it is modeled as a network of two state processes, because of performance issues (Roscoe and Goldsmith 1997). For each fact, reachable by intruder, $ignorantof(f)$ represents an unknown fact f and $known(f)$ represents a known fact f. These processes carry out $infer$ actions, which uses deductions to figure out unknown information from known facts.

$SayKnown$ process does not make any inferences, but generates already known messages to be sent to the receiving agent. The $chase()$ function instructs the model checker to apply partial-order reduction (Clarke 1999) on the selected process.

Overall intruder process is (without renaming and irrelevant details):

$$Intruder_{FC} = chase(||_{f:LearnableFacts}\ ignorantof(f)) \backslash \{infer\}) ||| \ SayKnown$$

The intruder process is a generic intruder process which is applicable for most protocol models. (Roscoe and Goldsmith 1997, Ryan, et al. 2001). Moreover, in this study, $Intruder_F$ process is used, whose structure is the same as $Intruder_{FC}$ process but does not contain the $chase$ reduction function:

$$Intruder_F = (||_{f:LearnableFacts}\ ignorantof(f)) \backslash \{infer\}) ||| \ SayKnown$$

These definitions of intruder are able to perform all actions of intruder in Dolev / Yao Model. It can *overhear* messages using the $hear$ channel: The $hear$ channel allows $Intruder$ receiving the messages that are communicated on the network by honest agents. Then, it builds a closed set of intruder knowledge, including the new message. Delivery of messages is possible, because the $Intruder$ process can perform the trace

$\langle hear.m_1, say.m_2 \rangle$ where $m_1 = m_2$, thereby delivering the message to intended recipient.

Also, it is possible for this process to block (intercept) messages, because it can reach a trace like $\langle hear.m_1, hear.m_2 \rangle$. In this case, upon hearing message m_1, intruder may not choose to send a message through say channel (because of the non-deterministic composition) and synchronize on another $hear$ event. To f*orge (fake)* messages, $Intruder$ process keeps a set of messages that it has heard and can deduce new protocol messages. It can send any of the messages in its knowledge set using say channel.

4.5.4.3. The Complete Network

The network of the processes is defined as the parallel composition of $Send$, $Recv$ processes and intruder process. Let $Send$ and $Recv$ be processes representing honest agents, such that; $Send = Send_0(Alice)$, $Recv = Recv_0(Bob)$, where $Alice, Bob \in Agent$. An event in the form of $send.a.b.m$ symbolizes sending message m from agent a to agent b. The network model, including agent and intruder processes and the channels, is illustrated in Figure 4.8.

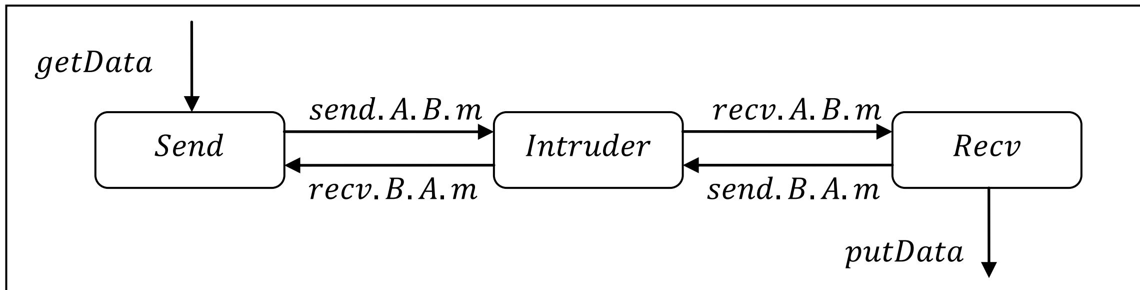

Figure 4.8. The Network Model Used for the Protocol

This model is defined a process composed of parallel composition of agent and intruder processes. The agent processes only communicate the signal event $test$ with each other and $send$ and $recv$ events to the intruder. Therefore, a $Network$ process is defined, expressing their relationship:

$$Network = (Send \underset{\alpha Test}{||} Recv) \underset{\alpha Stop}{||} STOP$$

where the sets $\alpha Test$ and $\alpha Stop$ are defined as: $\alpha Test = \{ test \}$, $\alpha Stop = \{ recv.Cameron \} \cup \{ send.a.Cameron \mid a \in Agent\}$. The purpose of the $\alpha Stop$ set is to prevent messages sent as and received by the agent Cameron. It is not a real agent, the intruder uses $Cameron$ only if it wants to impersonate a legitimate agent.

The $Intruder$ process is one of the $Intruder_B$, $Intruder_F$ or $Intruder_{FC}$ processes:

$$System = Network \underset{\alpha IA}{||} Intruder$$

where αIA is the synchronization set of parallel composition between the $Intruder$ process and the agent processes, such that $\alpha IA = \{ send, recv \}$.

$Send$ process only sends but does not receive any data and $Recv$ process does not send any data. However the $recv$ channel between $Send$ and $Intruder$ and the $send$ channel between $Recv$ and $Intruder$ are included in the model for the sake of generality.

4.5.5. Example Correct Run

In this section, the system's behavior during a correct run is evaluated. The $DataHash$ set defines a hash chain with a maximum length of 4; and $\langle d_0, d_1, d_1, d_0 \rangle$ be the data sequence.

Additionally, the $System_B$ process is used as defined in previous section, and the $Intruder_B$ process will be used as the intruder.

The run begins with the $Send_0(Alice)$ process choosing an agent process non-deterministically, which is the process Bob. It gets the data using $getData$ channel and sends it using the $send$ channel by producing the event $send.Alice.Bob.\langle d_0 \rangle$. $Intruder_Basic$ process synchronizes with this event and produces another event using the $recv$ channel, which is $recv.A.B.\langle d_0 \rangle$. $Recv_0(Bob)$ process synchronizes on this event. Then, sender again gets data, $\langle d_1 \rangle$, and calculates the hash of $\langle d_0 \rangle$, previous protocol message. This is accomplished by $hash$ function, performing a lookup from $DataHash$ set and returning the respective hash value: $\langle h_0 \rangle$. Thus, the 2nd message is sent, which is $send.A.B.\langle d_1, h_0 \rangle$.

To send the third message, $Send_1$ process chooses to get another data instead of sending the signature message and uses the previous two hash values, which were

passed as parameters to $Send_i$ process. Again, it prepares and sends the message via the $send$ channel, which is $send.A.B.\langle d_1, h_4, h_0\rangle$. Upon receiving this message, $Recv_i$ process adds this message to its buffer set R, which already contains first two messages. It already contains first two messages because, $Recv_0$ process has filled in these messages which were the parameters of $Recv_i$, after it had received second message.

The generation and sending of fourth message, which is $send.A.B.\langle d_0, h_{40}, h_4\rangle$ is similar to the third message. After the reception of fourth message, contents of the buffer set R becomes as follows:

$$R = \{(0, \langle d_0\rangle), (1, \langle d_1, h_0\rangle), (2, \langle d_1, h_4, h_0\rangle), (3, \langle d_0, h_{40}, h_4\rangle)\}$$

Finally, sender decides to send the signature message by evolving into $Send_n$ process. The signature message is composed of the hashes of third and fourth messages, encrypted with the private key of the sender: $send.A.B.Pk.(sk.A, \langle h_{40}, h_{522}\rangle)$. The receiver knows the dual of $sk.A$, which is $pk.A$ (i.e. public key of A), and is able to decrypt the contents of the message. As the receiver could decrypt the message using A's public key, it believes that message is valid, and puts it into verified set, which is the parameter of $Check$ process.

Up to this point, the sender and receiver have completed their protocol runs and sender is ready to start a new run. However, the receiver needs to verify the messages before it can continue; so it evolves into $Check$ process.

Initially, the $Check$ process has the parameters:

$R = \{(0, \langle d_0\rangle), (1, \langle d_1, h_0\rangle), (2, \langle d_1, h_4, h_0\rangle), (3, \langle d_0, h_{40}, h_4\rangle)\}$
$V = \{(4, \langle h_{40}, h_{522}\rangle)\}$
$i = 3$

For $i = 3$, the process finds that $P_4 \in V$, so it outputs d_0 using $putData$ channel, adds P_3 to V set and removes it from R. In the next iteration, $i = 2$, process also finds out that $P_3, P_4 \in V$ and continues until the index 0 is verified. At the end of the $Check$ process, the contents of R and V sets are:

$R = \{\}$
$V = \{(0, \langle d_0\rangle), (1, \langle d_1, h_0\rangle), (2, \langle d_1, h_4, h_0\rangle), (3, \langle d_0, h_{40}, h_4\rangle), (4, \langle h_{40}, h_{522}\rangle)\}$

Obviously, the set R be still empty, if some protocol messages were left unverified, but the V set would not contain all the received messages. Additionally, as

the process starts validation from the last protocol message, the data would be output with an inverted order on $putData$ channel, which is not a problem.

Finally, the $Check$ process evolves into $Recv_0$ process, forgetting all values in sets R and V, allowing a fresh protocol run with the sender process.

4.6. Reduction of Infinite State Model to Finite Model

The protocol model developed in Section 4.4 is not suitable for model checking, in which the state space is very large and cannot be verified by the finite computing resources. Thus, in this section, the sources of larger state space is identified, solutions are stated and implemented in the protocol model.

4.6.1. Analyzing the Infinite State CSP Model

In the previous section, the EMSS protocol hash chaining mechanism are modeled using CSP; by assigning each possible protocol message a unique hash value. While this approach can successfully model the protocol, problems arise when verifying the model using model checking.

The first problem is the infamous state-space explosion problem. When FDR tries to construct the state space of the system, a new state is generated for each distinct item in the $DataHash$ set. However, the cardinality of $DataHash$ set can be quite high, even if a small $AllData$ set is used. For example, for $|AllData| = 2$, there will be around 150 million different ways of constructing fifth protocol message, as given in Table 4.1.

Table 4.1. The Size of $DataHash$ Set for Two Data Values

i	0	1	2	3	4
$\lvert DataHash_i \rvert$	2	4	72	12168	$\sim 150\ M$

This exponential increase of the cardinality of the $DataHash$ set makes model checking infeasible beyond four messages, even if two distinct data values are used.

Another problem is the time complexity of the $hash$ function. As the $DataHash$ set grows bigger, it would take more time to select the correct hash value from the set, even if all the required states could be held in memory.

There are two kinds of reductions to solve state space explosion problem. First one is the application of state space compression techniques which the model checker provides. These are general techniques which are designed to be run on any LTS or process. The second kind is the simplification of the model, which is the approach described in the next sections.

Therefore it can be concluded that the bulk of the state space explosion is caused by the complexity of the $DataHash$ set and the hash chain model needs to be refined by abstracting unnecessary details away.

4.6.2. Revisions on the Data Model

As discussed in previous sections, the $AllData$ set contains m different data values, which symbolizes all the data values in the system. However, the actual contents of the data transferred do not have an effect on the behavior of the protocol.

Thus, one data item is sufficient to represent all data values in the protocol and the upper bound of the size of $AllData$ set should be one. This is actually dropping the set and choosing one symbol to represent all data items, instead.

To implement this revision, the $AllData$ (and $DATA$ set in implementation) is omitted. The data item could be symbolized by a fact $d.Agent$, where $Agent$ is the set of all agent identities. This change makes verification more convenient, as the origin of the data is more interesting than its value.

This revision is sound, because different values of data items are ignored by the processes. So, there is no point in having more than one data value.

Conversion of the model is actually an application of data independence on this protocol model. Note that all processes, using the $AllData$ set, satisfy the $NoEqT_T$ condition; that is there are no implicit or explicit equivalence tests between the members of $AllData$ set. Additionally, in the traces, it does not matter which member of $AllData$ set is recorded. Hence, $|AllData| = 1$ and the replacement of $AllData$ set by a single value is a sound reduction.

4.6.3. Bounding the Size of Hash Sets

In the hashing mechanism presented, the hash value of a given protocol message is calculated explicitly using the $hash$ function, which maps each protocol message to a single *correct* hash value. All other hash values would be *incorrect*, as there is one correct hash value.

This observation means that, while building the state space of the system, one value represents the correct hash value and all the remaining values represent incorrect hash values. If all incorrect hash values are symbolized as a single value, the state space would be reduced greatly (Figure 4.9).

Thus, two symbolic facts are declared; h_C, represents a correct hash value and h_I, represents an incorrect hash value, with respect to the corresponding previous message. In this case, a message $P_i = \langle d.A, h_C, h_I \rangle$ represents a message, carrying data from agent $Alice$ and correct hash value for P_{i-1} and an incorrect hash value for P_{i-2}.

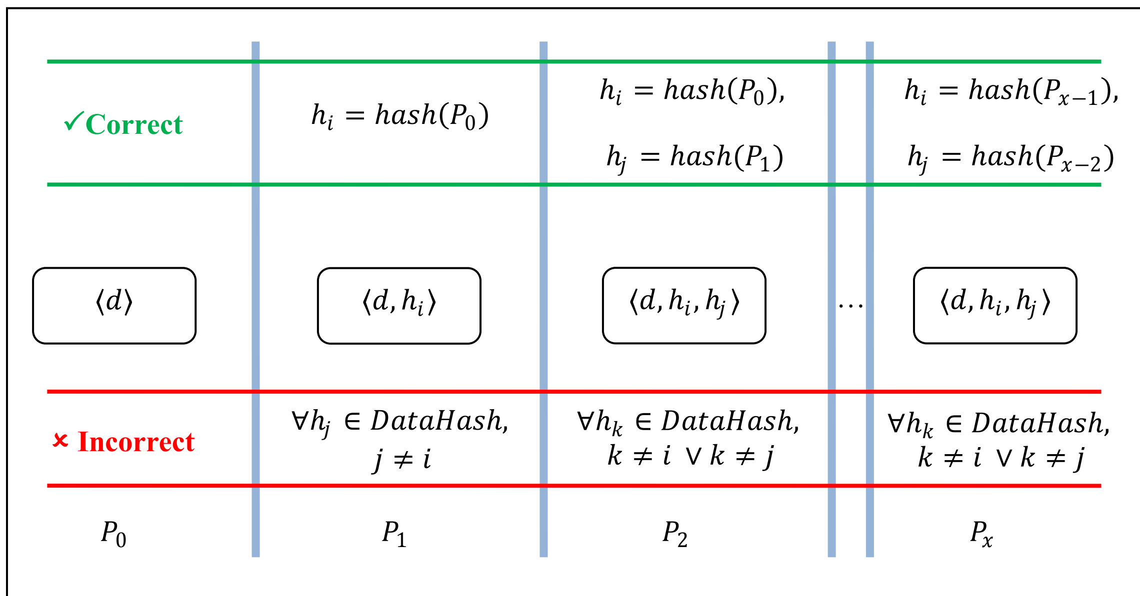

Figure 4.9. Correct and Incorrect Hash Values

Exactly one symbolic value is used instead of all of the incorrect hash values and there's no need to know which incorrect value has arrived. This approach greatly reduces the state space of the system and makes model checking possible, without losing any attacks.

For instance, the following set of messages represents a run of the protocol, without any intervention from the attacker:

$$\langle d.A\rangle, \langle d.A, h_C\rangle, \langle d.A, h_C, h_C\rangle, \dots, Pk.(sk.a, \langle h_C, h_C\rangle)$$

Next, $AllHashes$ and $DataHash$ sets need to be redefined. The new symbolic hash values h_C and h_I are members of $AllHashes$ such that $AllHashes = \{h_C, h_I\}$. Also, the $DataHash$ set no longer needs to contain all possible mappings of hashes and messages. It is redefined so that it contains only the messages whose hashes are correct:

$$DataHash = \{\langle data.A\rangle, \langle data.A, h_C\rangle, \langle data.A, h_C, h_C\ \rangle\}$$

The receiver needs to calculate the hashes of incoming messages, so a simplified hash function is used:

$$hash(ds) = if\ (h = \emptyset)\ then\ h_I\ else\ h_C$$

where $h = \{\ h \mid ds \in DataHash, ds = s\ \}$.

Having defined a new hashing mechanism, it is necessary to show that this mechanism is compatible with the hashing assumptions. The revised model satisfies the first assumption, because no mechanism is present to deduce the input from hash values h_I and h_C.

As all incorrect hash values are mapped to a single symbolic value h_I, different input messages can have the same hash value. However, it is unnecessary to know the exact incorrect hash value, but rather whether the value is right or wrong. In this sense, it can be concluded that the model satisfies this condition.

4.6.4. Revisions on Agent Processes

In the $Send$ process, the $getData$ events and previous hash values in the parameters of $Send_i$ and $Send_n$ are removed. Since it is assumed that the sender correctly calculated the previous hashes, which are equal to h_C, these parameters are not necessary.

Moreover, the $Last$ process is defined to be used for validation and verification which defines the behavior of the system after the protocol run.

$Send_0(id) = \sqcap\ b{:}\ Agent\ \bullet$

$\quad send.id.b.data.id \rightarrow$

$send.id.b.Sq.\langle data.id, h_C\rangle \rightarrow Send_i(id, b)$

$Send_i(id, b) = Send_n(id, b) \sqcap (send.id.b.Sq.\langle data.id, h_C, h_C\rangle \rightarrow$
$Send_{i+1}(id, b))$

$Send_n(id, b) = send.id.b.Pk.(sk.id, Sq.\langle h_C, h_C\rangle) \rightarrow Last$

The $hash$ function is not used in the $Send$ process, which reduces the time complexity of computing the transitions between states.

Similarly, the receiver process is modified to accommodate the changes, by dropping of all instances of $AllData$ set.

$Recv_0(id) = \square\, a: Agent \bullet$
$recv.a.id.d.a \rightarrow$
$\square\, hr_0: AllHashes \bullet$
$recv.a.id.Sq.\langle d.a, hr_0\rangle \rightarrow$
$Resp_i(id, a, \{(0, \langle d.a\rangle), (1, \langle d.a, hr_0\rangle)\})$

$Recv_i(id, a, Recvd) = Recv_n(id, a, Recvd) \square ($
$\square\, hr_1, hr_2 : AllHashes \bullet$
$recv.a.id.Sq.\langle d.a, hr_1, hr_2\rangle \rightarrow$
$Recv_{i+1}(id, a, Recvd \cup \{(i, \langle d.a, hr_1, hr_2\rangle)\})$

$Recv_n(id, a, Recvd) = \square\, hr_0, hr_1 : AllHashes \bullet$
$recv.a.id.Pk.(sk.a, Sq.\langle hr_0, hr_1\rangle) \rightarrow$
$Check(Recvd, \{(n, \langle hr_0, hr_1\rangle)\}, n - 1)$

In the $Check$ process, the hash of current message does not need to be compared with the hash values in previous messages. It is assumed that, if any previous message has been verified, then they contain the right hash value for the current message, which is h_C. Thus, it is needed to check whether any of the previous messages are in checked set and hash of the current message is correct.

Also, the recursion in $Check$ process is removed, so that the model stops if the protocol run is completed. Hence, $Recv_0$ is replaced by $Last$ process:

$Check(R, V, i) =$
$if\ i < 0\ then$
$Last$
$else\ if\ hash(P_i) = h_C\ and\ ((P_{i+1} \in V)\ or\ (P_{i+2} \in V))\ then$

$putData.data(P_i) \rightarrow$

$Check(R - P_i, V \cup P_i, i - 1)$

$else$

$Check(R - P_i, V, i - 1)$

4.6.5. Example Run using the Revised Model

Consider the example correct protocol run defined in Section 4.5.5. which used the infinite state model. The receiver's buffer set R containing the messages, before the reception of signature message were:

$$R = \{(0, \langle d_0 \rangle), (1, \langle d_1, h_0 \rangle), (2, \langle d_1, h_4, h_0 \rangle), (3, \langle d_0, h_{40}, h_4 \rangle)\}$$

Using the revised model, the sender agent sends $d.A$ as data value instead of $\{d_0, d_1\}$. Thus, from verification point of view, the distinction between different data values are eliminated, which were unnecessary. Additionally, as all the hash values are correct, they are represented using h_C in the protocol messages. Thus, the receivers' buffer set R' in revised model would be:

$$R' = \{(0, \langle d.A \rangle), (1, \langle d.A, h_C \rangle), (2, \langle d.A, h_C, h_C \rangle), (3, \langle d.A, h_C, h_C \rangle)\}$$

R' represents not only one correct run of the protocol which sends the data values $\langle d_0, d_1, d_1, d_0 \rangle$, but for all instances of data values, for a chain length of 4. That is, R' in revised model also represents a correct run, in which data values $\langle d_1, d_0, d_0, d_1 \rangle$ are sent.

Now, assume that the intruder modifies R, such that it replaces the hash value in message 1, such that message 1 becomes $\langle d_1, h_3 \rangle$. The h_3 value is not a correct hash value, and represented by h_I in the revised model. Hence, the corresponding message in revised model would be $\langle d.A, h_I \rangle$. Also, the hash of this message is h_I, so the $Check$ process does not verify this message.

Additionally, intruder might modify the data value, leaving the hash values intact. For example, the second protocol message in R can be changed to $\langle d_0, h_4, h_0 \rangle$. In the revised model, the changed message is symbolized by $\langle d.I, h_C, h_C \rangle$, in which the intruder has changed the data value $d.A$ with its own data value $d.I$. Again, the $Check$ process does not verify this message because its hash value evaluates to h_I.

4.7. Labeling the Sequence of Messages

The EMSS protocol definition does not specify a mechanism from which the receiver can know the sequence of the messages. The sequence numbers of messages are important for the $Check$ process, since it uses the sequence numbers to look them up from the set of received messages. Therefore, it is assumed that the sequence number of every message is sent along with the message in a way that the intruder is unable to learn or change. Intruder is still able to change the sequence of messages, but it cannot change the sequence numbers of the messages directly. The sequence labels used are elements of $LABEL_ALL$ where $LABEL_ALL = \{0..n\}$, n is the maximum chain length.

To illustrate the consequences of this problem and how the model used in this study deals with the problem; assume the message sequence sent by the $Send$ process:

$$\langle d.A\rangle, \langle d.A, h_C\rangle, \langle d.A, h_C, h_C\rangle, \langle d.A, h_C, h_C\rangle, Pk.(sk.a, \langle h_C, h_C\rangle)$$

If this message is intercepted by the attacker, $Recv$ process cannot get the third message. Then, the receiver forms the set R as:

$$R = \{(0, \langle d.A\rangle), (1, \langle d.A, h_C\rangle), (2, \langle d.A, h_C, h_C\rangle)\}$$

where, it would not notice the lost packet. However, it can be assumed that a lost message is equivalent to a message in which all fields contain garbage values:

$$R = \{(0, \langle d.A\rangle), (1, \langle d.A, h_C\rangle), (2, \langle d.A, h_C, h_C\rangle), (3, \langle d.I, h_I, h_I\rangle\}$$

This trace is also a part of state space, which in turn means that the revised model can handle lost messages as well.

CHAPTER 5

SPECIFICATION AND VERIFICATION OF THE PROPOSED PROTOCOL MODEL

In this chapter, specification processes are constructed for the validation and verification of the protocol model. The security properties of the protocol are formalized as CSP processes, which constitutes the specification of the protocol model. Validating the model means that checking whether the right model is built, on the other hand, verifying the model means that checking whether the model works correctly. Finally, refinement checking is applied on these processes to check whether the protocol adheres to the goals it claims to have.

Additionally an evaluation of the protocol model is presented, which includes the problems on the protocol itself and the considerations about the proposed models.

5.1. Parameters and Configurations Used

The proposed model is checked using the validation and verification properties stated in previous sections. FDR version 2.83 is used as the model checker on a system running Ubuntu 9.10 on Intel Core2 Duo $P7350$, 2.00 GHz processor and $4GB$ of RAM.

Both GUI and batch mode is present in FDR. GUI is used in the development phase of the CSP model and the batch interface is used to run the refinement checks found in this section.

In this study, refinement checks are run from batch interface of FDR. The command to start the check is:

$$./fdr2\ batch\ [OPTIONS] < input\ file >$$

The first parameter, $batch$, is used to notify FDR to run in batch mode to run all the checks in the specified $.csp$ file. The $OPTIONS$ parameter controls trace generation. Actual values of $OPTIONS$ parameter used in this study are:

- -$traces$: Instructs FDR to generate counter-example traces for any unsatisfied refinement checks. The generated traces are output to screen.
- -max 100: Maximum number of counter-example traces to generate. In this study, this value is set to 100. This value needs to be increased when maximum debug traces is reached in any debug trace.
- -$depth$ 2: This option instructs FDR to report trace information for sub-processes. In the refinement checks of this study, the traces of the top level processes are hidden (i.e. all τ actions). A depth level of 2 is required to be able to see τ actions.

The model and the refinement checks are both placed in $emss.csp$ file, which is the last parameter of the command. The other parameters of the model checker are left without any changes.

5.2. Specifications of the Proposed Model

In this section, the specification processes are built, which are used to check the proposed protocol model for errors. Validation specifications are used to check whether the process agents behave as defined in the protocol description. On the other hand, verification specifications focus on the authentication properties of the protocol.

Three system processes are used in specification properties. The first process is $System_B$ process, which uses $Intruder_B$ process is used as the intruder model.

$$System_B = Network \mid\mid Intruder_B$$

The second system process uses the full intruder without the $chase$ operation:

$$System_F = Network \mid\mid Intruder_F$$

The final system process uses the full intruder, which uses the $chase$ operation:

$$System_{FC} = Network \mid\mid Intruder_{FC}$$

Different system processes have only the $Intruder$ processes different. Other processes use the same definitions in all three system processes.

5.2.1. Validation Specifications

It is necessary to state some correctness properties of the protocol in order to validate the revised protocol model. For both properties, the $Last$ process is defined as $Last\ = test.ok \rightarrow STOP$.

Property 1: The first validation property checks whether the agents can finish the protocol run in a trusted network. To check the first property, two refinement checks are constructed. First refinement check asserts that a state where both agents perform the $test.ok$ event is reachable, using $System_B$:

$$STOP \sqsubseteq_T System_B \setminus \{\Sigma - \{test\}\}$$

where Σ is the set of all events in the system. If the model is correct, then this check should fail and return the debug trace of a correct run.

The second refinement check asserts the same property, using a specification process $Spec_0$, which is defined as $Spec_0 = test.ok \rightarrow STOP$. According to $Spec_0$, the system should eventually perform a $test.ok$ event. The refinement check is:

$$Spec_0 \sqsubseteq_T System_B \setminus \{\ \Sigma - \{test\}\ \}$$

This check should succeed if the system is correct; since the system's every behavior should be a subset of $Spec_0$'s behavior.

Property 2: The second validation property tests the protocol model using the full intruder model. So, $System_F$ and $System_{FC}$ processes are used as system processes in the refinement checks. The first and second refinement checks assert that a state where both agents perform the $test.ok$ event is reachable, using $System_F$ and $System_{FC}$ processes, respectively.

$$STOP \sqsubseteq_T System_F \setminus \{\Sigma - \{test\}\}$$
$$STOP \sqsubseteq_T System_{FC} \setminus \{\Sigma - \{test\}\}$$

If the proposed model is correct, the $test.ok$ event is reachable and both checks should return the debug trace of a successful protocol run.

Alternately, a specification process which shows the correct run of the system could be defined, in which $test$ and $putData$ events are observed:

$$Spec_1 =\ test.ok \rightarrow STOP \sqcap (\ \sqcap\ i: LABEL_ALL\ \bullet\ putData.i.Alice \rightarrow\ Spec_1)$$

According to $Spec_1$, the system process can communicate $test.ok$ without any data output or after output of some data. The refinement checks involving $Spec_1$ are:

$$Spec_1 \sqsubseteq_T System_F \setminus \{\Sigma - \{test, putData\}\}$$
$$Spec_1 \sqsubseteq_T System_{FC} \setminus \{\Sigma - \{test, putData\}\}$$

Both checks should fail, if the proposed model is correct, returning the trace of correct runs.

Property 3: The third (and last) property focuses on the order of $putData$ events, appear on the traces of the system. In a correct model, the sequence labels of two subsequent $putData$ events should be different and the second event should have a smaller value. To check this property, $Spec_2$ process is used:

$$Spec_2(i) = test.ok \rightarrow STOP \sqcap (\sqcap i_ : LABEL_ALL \bullet$$
$$i_ < i \;\&\; putData.i_.Alice \rightarrow Spec_2(i_)$$

The related refinement checks are:

$$Spec_2(n) \sqsubseteq_T System_F \setminus \{\Sigma - \{test, putData\}\}$$
$$Spec_2(n) \sqsubseteq_T System_{FC} \setminus \{\Sigma - \{test, putData\}\}$$

where n is the maximum hash chain length. The models should satisfy these checks to be correct.

5.2.2. Verification Specifications

Verification specifications state the specifications of the protocol, which is used in the checking of the model for potential attacks. In this study, only the one-way authentication property of EMSS protocol is verified.

If there is an attack on the protocol, then the intruder is able to trick the receiver to think that an invalid message is valid and other messages appear on the $putData$ channel, that are not sent by $Alice$. Therefore, the refinement checks based on this observation are:

$$STOP \sqsubseteq_T System_{FC} \setminus \{\Sigma - \{putData.i.Alice \mid 0 \leq i < n\}\}$$
$$STOP \sqsubseteq_T System_F \setminus \{\Sigma - \{putData.i.Alice \mid 0 \leq i < n\}\}$$

As the receiver process puts validated messages on $putData$ channel, any message that is not coming from $Alice$ is considered fake. Thus, if the protocol model has any attacks, this refinement check is not satisfied, and returning the debug trace of the attack. Otherwise, the check is satisfied.

If these refinement checks fail, debug traces will be generated, which show the sequence of messages forming the attack. On the other hand, if they are satisfied, it can be concluded that there are no attacks on this model of the protocol.

Both $System_F$ and $System_{FC}$ processes are used in this property. Although it is enough to apply refinement checking to one of these processes, both processes are checked.

5.3. Refinement Checking of the Proposed Model

The proposed model has been checked using the refinement checks defined in previous section. Table 5.1 shows the results of validation specifications.

Table 5.1. Results of Refinement Checks of Validation Properties

#	Property - Check	Intruder	# of States	# of Transitions	Result	# of counter examples
1	Property 1 – 1	$Intruder_B$	47	60	fail	1
2	Property 1 – 2	$Intruder_B$	48	61	pass	-
3	Property 2 – 1	$Intruder_F$	10,514	42,709	fail	32
4	Property 2 – 2	$Intruder_{FC}$	830	1,916	fail	1
5	Property 2 – 3	$Intruder_F$	10,546	42,901	pass	-
6	Property 2 – 4	$Intruder_{FC}$	831	1,917	pass	-
7	Property 3 – 1	$Intruder_F$	10,875	43,815	pass	-
8	Property 3 – 2	$Intruder_{FC}$	858	1,944	pass	-

The first property involved checking the agents' behavior using a basic intruder process. The first check failed and returned the debug trace of a correct protocol run, which is given in Figure 5.1. The second check was passed, which indicates the systems' behavior is equivalent to the $Spec_0$ process in Traces model.

$_tau$
$send.Alice.Bob.d.Alice$
$recv.Alice.Bob.d.Alice$
$send.Alice.Bob.Sq.< d.Alice, hC >$
$recv.Alice.Bob.Sq.< d.Alice, hC >$
$_tau$
$send.Alice.Bob.Pk.(sk.Alice, < hC, hC >)$
$recv.Alice.Bob.Pk.(sk.Alice, < hC, hC >)$
$putData.1.Alice$
$putData.0.Alice$
$test.ok$

Figure 5.1. The Debug Trace of the First Validation Property

The second property checks the model using $Intruder_F$ and $Intruder_{FC}$ processes. Checks one and two failed as expected, but there were different number of debug traces generated. This is an effect of the $chase$ function, in the $Intruder_{FC}$ process, which causes the model checker to follow immediate τ events. So, the state space is reduced and a shorter run than the trace in Figure 5.1 is returned (Figure 5.2).

$_tau$
$send.Alice.Bob.d.Alice$
$recv.Alice.Bob.d.Alice$
$recv.Alice.Bob.Sq.< d.Alice, hI >$
$send.Alice.Bob.Sq.< d.Alice, hC >$
$_tau$
$send.Alice.Bob.Pk.(sk.Alice, < hC, hC >)$
$recv.Alice.Bob.Pk.(sk.Alice, < hC, hC >)$
$putData.0.Alice$
$test.ok$

Figure 5.2. The Debug Trace for Second Property, using $Intruder_{FC}$

Additionally, the debug traces generated from the first check of second property are all possible correct runs, which are reachable in the state space of the model. The

results of the checks of second property point to a correct protocol model, as they are consistent with the expectations stated in previous section.

The checks for the last property were also satisfied, which was consistent with the expectations. Therefore, it is concluded that the model is valid and further verification can be performed on the model.

Verification properties were also performed using $Intruder_F$ and $Intruder_{FC}$ processes (Table 5.2.). All authentication properties are satisfied, so no attacks could be found on the proposed model.

Table 5.2. Results of Refinement Checks for Verification Properties

#	Property – Check	Intruder	# of States	# of Transitions	Result	# of counter examples
1	Property 1 – 1	$Intruder_F$	10,546	42,901	pass	-
2	Property 1 – 2	$Intruder_{FC}$	831	1917	pass	-

Analysis in previous studies could not find any attacks on protocol models of EMSS Protocol. Verification of EMSS Protocol using compositional proof rules in Team Automata also did not find any attacks on the protocol (ter Beek, et al. 2003, 2006)., where the protocol is verified with multiple senders and multiple receivers.

On the other hand, this study only uses single sender and single receiver. The limitation in this study could be removed using data independence techniques. However, only the application of data independence techniques is not enough. The protocol does not specify a way of distinguishing different message streams on the same network. Such a distinction is necessary for further analysis of the protocol.

Another verification of the EMSS protocol expressed the hash chains as graphs and reduced the authentication problem to the reachability problem over the graph (Perrig, et al. 2000). Although this study also proved the security properties of the protocol, the approach taken cannot be applied for general protocol schemas

CHAPTER 6

CONCLUSIONS

The main contribution of this study is the verification of EMSS authentication protocol by model checking using CSP / FDR framework. The EMSS protocol had been verified previously by only theorem proving techniques and this work constitutes the first attempt to model EMSS using model checking.

First of all, the infinite state protocol model was constructed based on the modeling assumptions presented in Section 4.2. Fixed sized hash value approach was introduced along with the necessary structures to support its operation, since the straightforward use of tagging cannot model hash chains in EMSS protocol model.

Although the infinite state model was successful in representing EMSS protocol in CSP, it was not suitable for symbolic model checking. Therefore, this model was revised and the hash chain model was identified as the source of the state space explosion problem. Moreover, the $hash$ function needed to lookup hash values from a very large $DataHash$ set which causes high time complexity of computing the transitions.

The application of revisions in Section 4.6 reduced the state space of the model to finite number of states. First of all, the size of $AllData$ set was limited to a single symbolic value which could symbolize all data values. Next, the hash chain model was revised. The hash values were subscripted in the infinite state model, all of which were used to denote correct and incorrect hash values. Therefore, h_C and h_I symbols are used to symbolize correct and incorrect hash values, which effectively reduced the state space.

After the application of aforementioned revisions, the revised model was validated and verified using FDR. The model was validated to check that the agent processes could function as described in the protocol definition. After validation, the model was verified for any attacks, but FDR could not find any attacks on the reduced protocol model.

This is the second study to model and verify a stream authentication protocol using CSP and model checking. The first study was the analysis of TESLA protocol using CSP and FDR (Broadfoot and Lowe 2002). This study is different in terms of modeling hash chains mechanism of the EMSS; since TESLA protocol used key chaining mechanisms. Therefore the CSP model of TESLA protocol requires more complex implementation of data independence techniques.

In addition to this study, the previous studies (ter Beek, et al. 2003, 2006) could not find any attacks on the EMSS protocol, using compositional proof rules of Team Automata. In contrast with this study, the protocol is proved to be correct for multiple sender and receivers scenario in these studies.

Future studies might include a generalized approach in modeling the stream authentication protocols, which roots from the comparison and contrasting with Lowe's work. This may also lead to tool like Casper (Lowe 1998) or an extension to Casper, which can allow automated checking of stream authentication protocols.

This work may also provide insight about different assumptions on the network (e.g. the receivers cannot communicate with each other, the intruder cannot block messages, etc…). For instance, weaker intruder models than Dolev - Yao Intruder model (Nguyen and Roscoe 2009) could be used as the intruder in the protocol model. Weaker intruders should cause a reduction in state-space, allowing for more complex behaviors of the protocol to be verified without losing any attacks.

REFERENCES

Abadi, M. (2000). "Security Protocols and Their Properties", *NATO ASI Series F Computer and Systems Sciences 175*: 39-60.

Alur, R. and Dill, D.L. (1994). "A Theory of Timed Automata", *Theoretical Computer Science 126*: 183-235.

Anderson, R., Bergadano, F., Crispo, B., Lee, J.H., Manifavas, C. and Needham, R. (1998). "A New Family of Authentication Protocols", *ACM SIGOPS Operating Systems Review 32*: 9-20.

Bolton, C. and Lowe, G. (2003). "On the Automatic Verification of Non-Standard Measures of Consistency", *Proceedings of 6th International Workshop on Formal Methods (IWFM).*

Bolton, C. and Lowe, G. (2004). "A Hierarchy of Failures-Based Models", *Electronic Notes in Theoretical Computer Science 96, Elsevier*: 129-152.

Bowman, H. and Gomez, R. (2006). Concurrency Theory, Calculi and Automata for Modelling Untimed and Timed Concurrent Systems, *Springer, ISBN-13: 9781852338954*.

Broadfoot, P.J. and Lowe, G. (2002). "Analysing a Stream Authentication Protocol Using Model Checking", *Lecture Notes In Computer Science 2502, Springer and Proceedings of the 7th European Symposium on Research in Computer Security*: 146-161.

Brookes, S.D., Hoare, C.A.R. and Roscoe, A.W. (1984). "A Theory of Communicating Sequential Processes", *Journal of the ACM 31*: 560-599.

Burrows, M., Abadi, M. and Needham, R. (1990). "A Logic of Authentication", *ACM Transactions on Computer Systems 8*: 18-36.

Canetti, R., Garay, J., Itkis, G., Micciancio, D., Naor, M. and Pinkas, B. (1999). "Multicast Security: A Taxonomy and Some Efficient Constructions", *18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM) 2*: 708-716.

Clarke E.M., Grumberg O. and Peled D.A. (1999). Model checking, *The MIT Press, ISBN-13: 978-0262032704*.

Creese, S.J. and Reed, J. (1999). "Verifying End-to-End Protocols Using Induction with CSP/FDR", *Lecture Notes in Computer Science 1586, Springer and Proceedings of IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International*

Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing: 1243-1257.

Donovan, B., Norris, P. and Lowe, G. (1999). "Analyzing a Library of Security Protocols Using Casper and FDR", *Proceedings of the Workshop on Formal Methods and Security Protocols*.

Davies, J. (2006). "Using CSP", *Refinement Techniques in Software Engineering, Springer*: 64-122.

Dilloway, C. and Lowe, G. (2007). "On the Specification of Secure Channels", *Proceedings of the Workshop on Issues in the Theory of Security (WITS '07)*.

Dolev, D. and Yao, A.C. (1981). "On the Security of Public Key Protocols", *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science, IEEE Computer Society*: 350-357.

Emerson, E.A. and Clarke, E.M. (1982). "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons", *Science of Computer Programming 2*: 241-266.

Gennaro, R. and Rohatgi, P. (1997). "How to Sign Digital Streams", *Lecture Notes in Computer Science 1297, Springer*: 180-197.

Glabbeek, R.J. van. (2000). "The Linear Time-Branching Time Spectrum I – The Semantics of Concrete, Sequential Processes", *Handbook of Process Algebra, Elsevier*: 3-99.

Goldsmith, M. (2005). "FDR2 User's Manual version 2.82" *Technical Report*.

Goldsmith, M. (2005). "Operational Semantics for Fun and Profit", *Lecture Notes in Computer Science 3525/2005, Springer*: 265-274.

Gorrieri, R., Martinelli, F. and Petrocchi, M. (2008). "Formal Models and Analysis of Secure Multicast in Wired and Wireless Networks", *Journal of Automated Reasoning 41, Springer*: 325-364.

Hoare, C.A.R. (1978). "Communicating Sequential Processes", *Communications of the ACM 21*: 666-677.

Hoare, C.A.R. (1985). Communicating Sequential Processes. *Prentice-Hall, Inc. International Series in Computer Science, ISBN-10:0131532715*.

Hui, M.L. and Lowe, G. (1999). "Safe Simplifying Transformations for Security Protocols or not just the Needham Schroeder Public Key Protocol", *Proceedings of the 12th IEEE workshop on Computer Security Foundations*: 32-43.

Hui, M.L. and Lowe, G. (2001). "Fault-Preserving Simplifying Transformations for Security Protocols", *Journal of Computer Security 9*: 3-46.

Keller, R.M., (1976). "Formal Verification of Parallel Programs", *Communications of the ACM 19/7*: 371–384.

Krzysztof, R. and Kozen, D.C. (1986). "Limits for Automatic Verification of Finite-State Concurrent Systems", *Information Processing Letters 22, Elsevier*: 307-309.

Lazic, R.S. (1998). "A Semantic Study of Data Independence with Applications to Model Checking", *Ph.D. Thesis, University of Oxford.*

Leuschel, M., Massart, T. and Currie, A. (2001). "How to Make FDR Spin LTL Model Checking of CSP by Refinement", *Lecture Notes in Computer Science 2021, Springer and Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*: 99 – 118.

Lowe, G. (1997). "A Family of Attacks Upon Authentication Protocols", *Technical Report, University of Leicester*.

Lowe, G. (1997). "A Hierarchy of Authentication Specifications", *Proceedings of the 10th IEEE workshop on Computer Security Foundations*: 31.

Lowe, G. (1997). "Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR", *Software – Concepts & Tools 17, Springer*: 93-102.

Lowe, G. (1998). "Casper: A Compiler for the Analysis of Security Protocols", *Journal of Computer Security 6*: 53-84.

Lowe, G. and Roscoe, A.W. (1997). "Using CSP to Detect Errors in the TMN Protocol", *IEEE Transactions on Software Engineering 23*: 659-669.

Mazurkiewicz, A. (1988). "Basic Notions of Trace Theory", *Lecture Notes in Computer Science 354, Springer*: 285–363.

Martinelli, F., Petrocchi, M. and Vaccarelli, A. (2003). "Analysing EMSS with Compositional Proof Rules for Non-Interference", *Proceedings of the 3rd Workshop on Issues in the Theory of Security*: 52-53.

Milner, R. (1982). A Calculus of Communicating Systems. *Springer-Verlag. ISBN: 0387102353*.

Needham, R.M. and Schroeder, M.D. (1978). "Using Encryption for Authentication in Large Networks of Computers", *Communications of the ACM 21/12*: 993 – 999.

Nguyen, L.H. and Roscoe, A.W. (2010). "Authentication Protocols Based on Low-Bandwith Unspoofable Channels: A Comparative Survey", *Journal of Computer Security (to be published)*.

Obraczka, K. (1998). "Multicast Transport Protocols: a Survey and Taxonomy", *IEEE Communications Magazine 36*: 94-102.

Ouaknine, J. and Worrell, J. (2002). "Timed CSP = Closed Timed Automata", *Electronic Notes in Theoretical Computer Science 68, Elsevier*: 142-159.

Perrig, A., Canetti, R., Tygar, J.D. and Dawn, S. (2000). "Efficient Authentication and Signing of Multicast Streams Over Lossy Channels", *Proceedings of the 2000 IEEE Symposium on Security and Privacy*: 56-75.

Petri, C.A. (1962). "Fundamentals of a Theory of Asynchronous Information Flow", *Proceedings of the International Federation for Information Processing (IFIP) Congress 62*: 386-390.

Pnueli A. (1977). "The Temporal Logic of Programs", *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*: 46-77.

Reed, G.M. and Roscoe, A.W. (1988) "A Timed Model for Communicating Sequential Processes", *Theoretical Computer Science 58, Elsevier*: 249-261.

Roscoe, A.W. (1995). "Modelling and Verifying Key-Exchange Protocols Using CSP and FDR", *Proceedings of 8th IEEE workshop on Computer Security Foundations*: 98-107.

Roscoe, A.W. (1995). "CSP and Determinism in Security Modelling", *Proceedings of 1995 IEEE Symposium on Security and Privacy*: 114.

Roscoe, A.W. (1996). "Intensional Specifications of Security Protocols", *Proceedings of the 9th IEEE workshop on Computer Security Foundations*: 28.

Roscoe, A.W. (1998). The Theory and Practice of Concurrency, *Prentice Hall, Inc. ISBN-13: 978-0136744092.*

Roscoe, A.W. (2005). "On the Expressive Power of CSP Refinement", *Formal Aspects of Computing 17, Springer*: 93-112.

Roscoe, A.W. (2008). "The Three Platonic Models of Divergence-Strict CSP", *Lecture Notes in Computer Science 5160, Springer and Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC)*: 23-49.

Roscoe, A.W. (2009). "Revivals, Stuckness and the Hierarchy of CSP Models", *Journal of Logic and Algebraic Programming, ScienceDirect 78/3:* 163-190.

Roscoe, A.W. and Broadfoot, P.J. (1999). "Proving Security Protocols with Model Checkers by Data Independence Techniques", *Journal of Computer Security 7*: 147-190.

Roscoe, A.W. and Broadfoot, P.J. (2002). "Internalising Agents in CSP Protocol Models", *Proceedings of 2nd Workshop on Issues in the Theory of Security*.

Roscoe, A.W., Broadfoot, P.J. and Lowe, G. (2000). "Automating Data Independence", *Lecture Notes in Computer Science 1895, Springer and Proceedings of the 6th European Symposium on Research in Computer Security*: 175-190.

Roscoe, A.W. and Goldsmith, M.H. (1997). "The Perfect Spy for Model-Checking Crypto-Protocols", *Proceedings of DIMACS Workshop on the Design and Formal Verification of Crypto-Protocols*.

Roscoe, A.W. and Kleiner, E. (2006). "Modelling Unbounded Parallel Sessions of Security Protocols in CSP" *Unpublished Draft*.

Roscoe, A.W., Gardiner, P. H. and Goldsmith, M.H. (1995). "Hierarchical Compression for Model-Checking CSP, or How to Check 10^20 Dining Philosophers for Deadlock", *Lecture Notes in Computer Science 1019 and Proceedings of the 1st International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*: 133 – 152.

Roscoe, A.W., Reed, J.N. and Sinclair, J.E. (2007). "Responsiveness and stable revivals", *Formal Aspects of Computing, Springer 19/3*: 303-319.

Ryan, P. and Schneider, S. (1998)."An Attack on a Recursive Authentication Protocol: A Cautionary Tale", *Information Processing Letters 65, Elsevier*: 7-10.

Ryan, P., Schneider, S., Goldsmith, M., Lowe G. and Roscoe, A.W. (2001). The Modelling and Analysis of Security Protocols: the CSP Approach. *Addison-Wesley Professional, ISBN-10: 0201674718*.

Scattergood, B. (1998). "The Semantics and Implementation of Machine-Readable CSP", *Ph.D. Thesis, University of Oxford*.

Shaikh, S.A., Bush, V.J. and Schneider, S. (2009). "Specifying Authentication Using Signal Events in CSP", *Computers & Security 28, Elsevier*: 310-324.

Schneider, S. (1996). "Security Properties and CSP", *Proceedings of the 1996 IEEE Symposium on Security and Privacy*: 174-189.

Schneider, S. (1997). "Verifying Authentication Protocols with CSP", *Proceedings of the 10th IEEE workshop on Computer Security Foundations*: 9.

Schneider, S. (1999). Concurrent and Real-Time Systems: the CSP Approach. *John Wiley. ISBN-13: 978-0471623731*.

Schneider, S. (2002). "Verifying Authentication Protocol Implementations", *Proceedings of 5th International Conference on Formal Methods for Open Object-Based Distributed Systems*: 5.

Schneider, S. and Delicata, R. (2005). "Verifying Security Protocols: An Application of CSP", *Lecture Notes in Computer Science 3525/2005, Springer*: 243-263.

Tarski, A. (1955). “A Lattice-theoretical Fixpoint Theorem and Its Applications”, Pacific Journal of Mathematics 5/2: 285–309.

ter Beek, M.H., Lenzini, G. and Petrocchi, M. (2003).“Team Automata for Security Analysis of Multicast/Broadcast Communication”, *Proceedings of Workshop on Issues in Security and Petri Nets*: 57-71.

ter Beek, M.H., Lenzini, G. and Petrocchi, M. (2006). “A Team Automaton Scenario for the Analysis of Security Properties of Communication Protocols”, *Journal of Automata, Languages and Combinatorics 11*: 345-374.

Winskel, G. and Nielsen, M. (1995). “Models for Concurrency”, *Handbook of Logic in Computer Science Semantic Modelling 4*: 1-148.

Wolter, U. (2002). “CSP, Partial Automata, and Coalgebras”, *Theoretical Computer Science 280, Elsevier*: 3-34.

APPENDIX A

```
-- -----------------------------------------------------------------
-- Data Types and Sets
-- -----------------------------------------------------------------
datatype fact =  Sq. Seq(fact) | Hash.fact | pk. AGENT |
                 sk. AGENT | Pk. (fact, Seq(fact)) |
                 Alice | Bob | Cameron | hC | hI | d.AGENT

datatype TEST_ = ok

AGENT = {Alice, Bob, Cameron}
HASH = {hC, hI}

DataHash = {<d.Alice>, <d.Alice,hC>, <d.Alice,hC,hC>}
hash(s) =  let
                hx = { hC | ds <- DataHash, ds == s }
           within
                if empty(hx) then
                     hI
                else
                     pick(hx)

n = 6
LABEL = {2..n-1}
LABEL_ALL = {0..n}

PM_0 = {  (0, d.a) | a <- AGENT }
PM_1 = {  (1, Sq.<d.a, h>) | a <- AGENT, h <- HASH }
PM_c = {  (c_, Sq.<d.a, h1, h2>) | a <- AGENT, h1 <- HASH,
          h2 <- HASH, c_ <- LABEL}
PM_n = {  (c_, Pk.(sk.a, <h1, h2>)) | a <-AGENT, h1 <- HASH,
          h2 <- HASH, c_ <- LABEL_ALL}

PM = Union({ PM_0, PM_1, PM_c, PM_n })
MESG_BODY = {m_ | (_,m_) <- PM}

Fact = Union({
      AGENT,
      {pk.a | a <- AGENT},
      {sk.a | a <- AGENT},
      HASH,
      {d.a | a <- AGENT},
      MESG_BODY
      })

channel send, recv : AGENT . AGENT . PM
channel putData : LABEL_ALL . AGENT

channel test : TEST_

-- -----------------------------------------------------------------
-- General Helper Functions
-- -----------------------------------------------------------------
```

```
dual(pk.arg) = sk.arg
dual(sk.arg) = pk.arg

second_((_,m_)) = m_
first_((i_,_)) = i_

data_( <d.a_> ) = a_
data_( <d.a_, _> ) = a_
data_( <d.a_, _, _> ) = a_

pick({x}) = x

isMember_(V, i) =
      if empty({ (id_, m_) | (id_, m_) <- V, id_ == i}) then
            false
      else
            true
-- -------------------------------------------------------------
-- Agent Processes
-- -------------------------------------------------------------

Sender_Alice = Sender_0(Alice)
Receiver_Bob = Recv_0(Bob)

-- Sender Process
-- Send protocol messages 1 and 2
Sender_0(id) = |~| b: diff(AGENT, {id}) @
            send.id.b.(0, d.id) ->
            send.id.b.(1, Sq.<d.id, hC>) ->
                        Sender_i(id,b, 2)

-- Send ith protocol message or the final message
Sender_i(id, b, i) = Sender_n(id, b, i)
                        |~| (
                        (i < n) &
                             send.id.b.(i, Sq.<d.id, hC, hC>) ->
                                        Sender_i(id, b, i+1) )

-- Send signature message and stop
Sender_n(id, b, i) = send.id.b.(i, Pk.(sk.id, <hC, hC>)) ->
                          Last

-- Receiver Process
-- Receive protocol messages 1 and 2
Recv_0(id) = [] a: diff(AGENT, {id}) @
                 recv.a.id.(0, d.a) ->
                 [] hr0 : HASH @
                 recv.a.id.(1, Sq.<d.a, hr0>) ->
                 Recv_i(id, a, {(0, <d.a>), (1, <d.a, hr0>)} , 2)

Recv_i(id, a, received, i) = Recv_n(id, a, received, i) []
      ( [] hr1 : HASH @
            [] hr2 : HASH @
                 (i < n) &
                        recv.a.id.(i, Sq.<d.a, hr1, hr2>) ->
```

```
     Recv_i(id, a, union(received, {(i, <d.a, hr1, hr2>)}),
i+1))

Recv_n(id, a, received, i) = [] hr1 : HASH @
                                [] hr2 : HASH @
          recv.a.id.(i, Pk.(sk.a, <hr1, hr2>)) ->
          Checker_i(received, {(i, <hr1, hr2>)}, (i-1))

Checker_i(R, V, i) =
     let
          self_ = { (id_, m_) | (id_, m_) <- R, id_ == i}
          self = if empty(self_) then
                    <>
                 else
                    second_(pick(self_))
     within
          if i < 0 then
               Last
          else
               if (hash(self) == hC) then
                    if (isMember_(V, i+1)
                         or isMember_(V, i+2)) then
                         putData.i.data_(self) ->
                              Checker_i(diff(R, self_),
                                   union(V, self_),  i-1)
                    else
                         Checker_i(diff(R, self_), V,  i-1)
               else
                    Checker_i(diff(R, self_), V,  i-1)

-- Last Process
Last = test.ok -> STOP

-- -----------------------------------------------------------
-- Intruder
-- -----------------------------------------------------------

-- Deductions
SqDeductions(X) = { (set(fs_), Sq.fs_) | Sq.fs_ <- X }

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

EncryptionDeductions(X) = { (union( {k_}, set(fs_) ),
                         Pk.(k_,fs_) ) | Pk.(k_, fs_) <- X }

DecryptionDeductions(X) = {({Pk.(k_,fs_), dual(k_)}, f_) |
                           Pk.(k_,fs_) <- X, f_ <- set(fs_)}

DeductionSet(X) = Union({SqDeductions(X), UnSqDeductions(X),
EncryptionDeductions(X), DecryptionDeductions(X)})

AllDeductions = DeductionSet(Fact)

Close(S) = let S' = {f | (X, f) <- AllDeductions, diff(X,S)=={}}
```

```
        within
      if diff(S',S)=={} then S else Close(union(S,S'))

-- Intruder Initial Knowledge
IK' = Union({
     AGENT,
     HASH,
     { pk.a | a <- AGENT},
     {sk.Cameron}
     })

IK = Close(IK')

PossibleBasicKnowledge = union(IK,MESG_BODY)
KnowableFacts = Close(PossibleBasicKnowledge)
LearnableFacts = diff(KnowableFacts,IK)
Deductions = {(X,f) | (X,f) <- AllDeductions,
                           member(f,LearnableFacts),
                           not member(f,X),
                           diff(X,KnowableFacts)=={}}

-- The Intruder Process

channel say,learn : MESG_BODY
channel infer : Deductions

ignorantof(f) = learn?f:MESG_BODY -> knows(f)
                [] infer?t:{(X,f') | (X,f') <- Deductions,
                     f'==f} -> knows(f)

knows(f) = say?f:MESG_BODY -> knows(f)
           [] learn?f:MESG_BODY -> knows(f)
           [] infer?t:{(X,f') | (X,f') <- Deductions,
                member(f,X)} -> knows(f)

AlphaL(f) = Union({{say.f,learn.f | member(f,MESG_BODY)},
                  {infer.(X,f') | (X,f') <- Deductions, (f' ==
                     f) or member(f,X)}})

SayKnown = say?f:inter(IK,MESG_BODY) -> SayKnown
           [] learn?f:inter(IK,MESG_BODY) -> SayKnown

transparent chase

IntruderC'' = chase((|| f:LearnableFacts @  [AlphaL(f)]
                                 ignorantof(f))\{|infer|})
Intruder'' = ((|| f:LearnableFacts @  [AlphaL(f)]
                                 ignorantof(f))\{|infer|})

IntruderC' = IntruderC'' ||| SayKnown
Intruder' = Intruder'' ||| SayKnown

Intruder_F = (Intruder'
                [[ learn.second_(m) <- send.A.B.m | A <- AGENT,
                           B <- AGENT, m <- PM ]]
```

```
                [[ say.second_(m) <- recv.A.B.m | A <- AGENT,
                           B <- AGENT, m <- PM ]]
              )

Intruder_FC = (IntruderC'
                [[ learn.second_(m) <- send.A.B.m | A <- AGENT,
                           B <- AGENT, m <- PM ]]
                [[ say.second_(m) <- recv.A.B.m | A <- AGENT,
                           B <- AGENT, m <- PM ]]
              )

-- Basic Intruder

Intruder_B = ([] a: AGENT, b: AGENT @
                [] m: PM @
                 send.a.b.m -> recv.a.b.m -> Intruder_B)

-- ---------------------------------------------------------------
-- System Process
-- ---------------------------------------------------------------
Network = (Sender_Alice [| {| test |} |] Receiver_Bob ) [|
                union({| recv.Cameron |}, {| send.a_.Cameron |
                           a_ <- AGENT |}) |] STOP

System_F = Network [| {| send, recv |} |] Intruder_F
System_FC = Network [| {| send, recv |} |] Intruder_FC
System_B = Network [| {| send, recv |} |] Intruder_B

-- ---------------------------------------------------------------
-- Specifications
-- ---------------------------------------------------------------

-- Validation
Spec0 = test.ok -> STOP

Spec1 = (|~| i: LABEL_ALL @
                putData.i.Alice -> Spec1) |~| test.ok -> STOP

Spec2(i) = (|~| i_:LABEL_ALL @
                       i_ < i & putData.i_.Alice -> Spec2(i_))
           |~| test.ok-> STOP

assert Spec0 [T= System_B \ {| send, recv, putData |}

assert Spec1 [T= System_F \ {| send, recv |}
assert Spec1 [T= System_FC \ {| send, recv |}

assert Spec2(n) [T= System_F \ {| send, recv |}
assert Spec2(n) [T= System_FC \ {| send, recv |}

assert STOP [T= System_F \ {| send, recv, putData |}
assert STOP [T= System_FC \ {| send, recv, putData |}

assert STOP [T= System_B \ {| send, recv, putData |}

-- Verification
```

```
assert STOP [T= System_F \ {| send, recv, test, putData.i_.a_ |
                                i_ <- LABEL_ALL,
                                a_ <- AGENT,
                                a_ == Alice |}

assert STOP [T= System_FC \ {| send, recv, test, putData.i_.a_ |
                                i_ <- LABEL_ALL,
                                a_ <- AGENT,
                                a_ == Alice |}
```