Directed Symbolic Model Checking of Security Protocols

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Qurat ul Ain Nizamani

Department of Computer Science

University of Leicester

2011

Declaration

The content of this submission was undertaken in the Department of Computer Science, University of Leicester, and supervised by Dr. Emilio Tuosto during the period of registration. I hereby declare that the materials of this submission have not previously been published for a degree or diploma at any other university or institute. All the materials submitted is the result of my own research except as cited in the references.

Preliminary versions of the results presented in this submission have been published in the following papers:

- Qurat ul Ain Nizamani and Hyder A. Nizamani. *Analysis of a Federated Identity Management Protocol in SOC. In Proceedings of the 3rd Young Researchers Workshop on Service Oriented Computing (YR-SOC), 2008.*
- Qurat ul Ain Nizamani and Emilio Tuosto. *Heuristic Methods for Security Protocols. In Proceedings of 7th International Workshop on Security Issues, EPTCS Volume 7, 2009. pp 61-75*

Abstract

This thesis promotes the use of *directed model checking* for security protocol verification. In particular, we investigated the possibility of designing heuristics that can reduce the overall size of the state space and can direct the search towards states containing an attack. More precisely,

- We have designed three property-specific heuristics namely $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$. The heuristics derive their hints from the security property to be verified and assign weights to states according to their possibility of leading to an attack.
- $\mathcal{H}_1$ is formally proved correct, i.e., the states pruned by the heuristic $\mathcal{H}_1$ do not contain any attack.
- An existing tool $\mathcal{A}$SPASyA with conventional model checking algorithm (i.e., depth first search) has been modified so as to integrate our heuristics into it. The resulting tool $\mathcal{H}$-ASPASyA uses an informed search algorithm that is equipped with our heuristics. The heuristics evaluate the states which are then explored in decreasing order of their weights.
- The new tool $\mathcal{H}$-ASPASyA is tested against a few protocols to gauge the performance of our heuristics.

The results demonstrate the efficiency of our approach. It is worth mentioning that despite being a widely applied verification technique, model checking suffers from the state space explosion problem. Recently directed model checking has been used to mitigate the state space explosion problem in general model checking. However, the directed model checking approaches have not been studied extensively for security protocol verification. This thesis demonstrates the fact that directed model checking can be adapted for security protocol verification in order to yield better results.

Acknowledgements

First and Foremost, my sincerest gartitude to my supervisor Dr. Emilio Tuosto. His guidance, teaching, and support helped me to complete my research within time. He devoted his precious time to teach me the principles of quality research, critical analysis, and to express my ideas clearly.

Many thanks to the teaching and administrative staff of the University of the Leicester, UK for providing a comfortable working environment. In particular, I am thankful to my second supervisor Prof. José Fiadeiro, Ph.D director Prof. Thomas Erlebrach, and Ph.D tutor Dr. Fer-Jan De Vries for their support and cooperation during my Ph.D studies.

Many thanks to all my friends who were always there to help me whenever I needed them.

Finally, I am indebted to my family for the love and prayers they always shower upon me. In particular, to my late father Ghulam Rabbani Agro, who gave me the gifts of enthusiasm, self confidence, and moral values. Baba! I owe all my success to you. I am also thankful to my mother, parents in-law, brothers, and uncles & aunties for their love, care, and prayers. Last but not least, my thanks to my husband Hyder Ali Nizamani, for being understanding and supportive and my son Talha Nizamani for being a bundle of joy during tough times of studies and coping with the times when I were not available for him.

Dedicated to Baba, with Love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The widespread use of distributed and network applications (such as e-banking, e-commerce, e-voting, etc.) require secure communications over an insecure network. Communicating parties therefore need to be assured that certain security properties are guaranteed during such conversations. For instance, they usually require that messages exchanged during communication are only accessible to honest participants, i.e., secrecy is guaranteed. They also desire to ascertain the identity of the party they are communicating with, that is authentication must be achieved. They may also expect certain other behaviours such as non-repudiation, integrity, reliability, etc.

The tools that ensure such guarantees are called security protocols. Such protocols rely on cryptographic primitives (mainly encryption, decryption, and hash functions) to achieve such goals. However, designing security protocol, is hard and typically error prone even in simple cases. A number of protocols that have been considered *secure* over a number of years have later proved to be flawed. One of the paramount examples is the Needham Schroeder public key protocol first proposed in [61] and discovered to be vulnerable in [55].

The inherent ambiguity in security protocols propelled the need to test security protocols extensively before they are put into operation. The formal verification community took up this challenge and many researchers have come up with various verification frameworks and tools employing model checking [49, 12], strand spaces [69, 68], path analysis [57, 1, 16, 17], theorem proving [64], to name a few. However, one of the interesting aspects in the analysis of security protocols is the complexity of verification algorithm. The security protocol verification problem is undecidable in general [60] and computationally hard [38, 27, 28, 67] in the rest of the cases.

Undecidability arises as the protocols are assumed to be operating in a network that is under the control of an intruder. The intruder is usually modelled as a powerful agent who has many capabilities. Many such intruder models are available, among which Dolev-Yao [37] is presumably the most powerful. The motivation behind using such a powerful intruder comes from the fact that a protocol withstanding an attack by a powerful intruder can resist a weaker intruder in reality. A Dolev-Yao intruder is capable of intercepting, replaying, inserting, and forging messages by pairing, encryption, and decryption. However, he can not perform cryptanalysis, that is perfect cryptography is assumed. The perfect encryption hypothesis states that an encrypted message can only be deciphered using appropriate keys and the only way to generate a cryptogram such as $\{m\}_k$ is that the intruder knows the key k. Thus verifying a protocol with any number of sessions that can be interleaved together and in the presence of a Dolev-Yao intruder is computationally hard.

Among the various verification approaches, *model checking* (MC) is the widely used and applied technique. In model checking, a protocol is analysed for all possible behaviours (under specified conditions), i.e., all states must be enumerated and checked to find the error states. However, the problem with MC is the exponential growth of the state space with the increase in number of system components. This is termed as *state space explosion*, i.e., system resources are exhausted before the verification algorithm terminates.

Many authors have suggested different techniques to tackle state space explosion; for instance, symbolic techniques [4, 18, 19, 43], partial order reduction [32, 44], abstractions [31], symmetries [42, 30], to name a few. Recently, the focus has been on *directed model checking* (DMC) [40, 53] which emerges as an amalgam of AI inspired heuristic search techniques and MC. The main idea is to help searching algorithms find error states quickly before it runs out of resources. More specifically, the searching algorithm is equipped with *hints* along with the description of the problem at hand, that prioritise states according to their proximity to goal states (e.g., states violating a correctness properties). However, its important to mention here that such techniques have not been keenly pursued in security protocol verification. In fact, there is very little evidence of exploiting heuristic techniques in security protocol analysis.

This thesis promotes the use of directed model checking for security protocol analysis. More precisely, we have identified three heuristics that can guide the searching algorithm towards attack states and establish a conclusion after exploring relatively fewer states. The effectiveness of such approach has been shown by integrating them into the symbolic model checker $\mathcal{A}$SPASyA [9]. It is worth mentioning that although the heuristics have been defined in terms of a formal framework [21], they are rather general and can be adopted by similar verification approaches.

1.2 Research Question

The huge state spaces generated while model checking is a main hindrance in the verification of large, complex, and realistic protocols. It is therefore important to investigate methods and techniques through which state space can be efficiently managed and explored. The following research question has been formulated in this context.

Is it possible to exploit domain knowledge to devise heuristics for model checking security protocols that can help the model checking algorithm to efficiently explore the state space without eliminating any attacks?

DMC can be a promising improvement to conventional model checking and is characterized by heuristic supported search techniques that explore the states according to their proximity to goal states. In terms of security protocols, the goal for searching algorithms is an attack on the protocol. Therefore, we have investigated DMC as a plausible technique for security protocol verification.

Note that our focus is on the design of heuristics and not on the heuristic search algorithms to be used in the DMC approach. Mainly, the heuristics used in directed model checking can contribute to efficiency by the following ways:

1. Reducing the overall size of the state space by discarding those states that can not lead to an attack. These states are never explored during the state space construction and this capability of heuristics is referred to as *pruning*.

2. Driving the search towards promising states and finding a counter example before the resources exhaust. We will refer this capability of heuristics as *re-ordering*.

3. Combining the above two approaches so that pruning and re-ordering both can contribute to efficiency.

A heuristic exhibiting re-ordering contributes to efficiency when the aim of the model checking algorithm is to find an attack on the protocol. More precisely, the attack state is discovered more quickly as the state space is not explored systematically in a pre-defined order e.g. *depth first search* (DFS) or *breadth first search* (BFS); in fact the order of exploration is determined by the weight of each state generated by the heuristic function. However, if the goal of model checking is to discover all attacks on the protocol, re-ordering renders no benefits. More precisely, visiting the state space in one order or another does not make any difference when complete state space is to be explored. Such situations demand heuristics with pruning capabilities as they reduce the overall size of the state space to be explored and significantly contribute to efficiency. Usually the aim of DMC is re-ordering [47], however, our focus is on designing heuristics that can preferably demonstrate both capabilities.

1.3 Methodology and Main Contributions

We address the research question stated in § 1.2 by dividing the task in three steps:

- developing the heuristics and formal proofs of their properties;
- integrating the heuristics into a MC algorithm; and
- testing the efficiency of the new algorithm.

In order to design the heuristics we use a framework [43], which represents the behaviour of a security protocol through *Cryptographic Interaction Pattern calculus* (*cIP*) and *protocol Logic* ($\mathcal{PL}$), respectively a cryptographic process calculus and a logic for specifying security properties. Note that heuristics are defined for the framework [43] but they can be easily adopted by other security protocol verification frameworks (see § 1.4.2). The heuristics are then implemented by integrating them into the symbolic model checker $\mathcal{A}$SPASyA [9, 43]. In particular, we adopted a hybrid model checking approach that combines directed and conventional MC. More precisely, the state space is initially explored by means of an informed search algorithm followed by a conventional DFS algorithm. Finally, the new algorithm called $\mathcal{H}$-ASPASyA is tested with a few protocols to observe the behaviour and respective efficiency of each heuristic.

The main contributions of this research are enumerated in the following paragraphs.

Heuristic Function $\mathcal{H}_1$ The heuristic $\mathcal{H}_1$ exploits $\mathcal{PL}$ formulae formalising the security properties of interest. Such heuristic may drive the search of an attack path and has the following capabilities:

- The heuristic ranks the nodes and the edges of the state space by inspecting (the syntactical structure of the) formula expressing the security property of interest. More precisely, the state space consists of the transition system representing the possible runs of a protocol; the heuristic weighs states and transitions considering the instances of principals that joined the context and how they are quantified in the security formula. Weights are designed so that most promising paths are tried before other less promising directions.
- The heuristic can rule out portions of the state space that does not contain attacks. As mentioned in § 1.2, such capability is beneficial when there is no attack on the protocol or when the verifier is interested in finding all attacks on the protocol. In both cases the searching algorithm needs to explore the complete state space in order to establish the correctness of the protocol under specified assumptions.

Heuristic Function $\mathcal{H}_2$ Another property dependent heuristic ($\mathcal{H}_2$) is defined. It exploits the atomic formulae and logical connectives in $\mathcal{PL}$ formulae to assign weights to states. Intuitively, $\mathcal{H}_2$ looks at the assignment of open variables in a security formula and checks the mappings in a state to mark the state as a promising or pruned one.

Heuristic Function $\mathcal{H}_3$ Heuristic function $\mathcal{H}_3$ has been designed to evaluate the combined effect of $\mathcal{H}_1$ and $\mathcal{H}_2$. The heuristic $\mathcal{H}_3$ is therefore a composite heuristic as it ranks states with the help of two different heuristic functions i.e., $\mathcal{H}_1$ and $\mathcal{H}_2$ whichever is applicable to the state.

Properties of the Heuristics An *estimation based* heuristic (see § 2.5.2.1) ranks nodes according to their cost of reaching the goal node. For such heuristics, it is typically required that *admissibility* and *consistency* or *monotonicity* [54] of the heuristics are also proved. Admissibility requires that heuristic estimates should never exceed the actual cost of reaching the goal node. Consistency demands that for a node n and its successor n' , the heuristic estimate for n to reach the goal node will be no more then the step cost of reaching the n' plus the heuristic estimate for n' to reach the goal node. However, both consistency and admissibility can not be applied on evaluation-based heuristics. Mainly due to the reason that an evaluation function ranks the nodes according to how *promising* is the state in terms of leading to a goal state. Since our heuristic function is an evaluation-based function, admissibility and consistency are not proved as it could make no sense. We discuss this issue in detail in § 3.5. For heuristic $\mathcal{H}_1$, the correctness of the heuristic, namely that no attacks can be found in the portion of the state space cut by our heuristic, is given in § 3.2.1.

Implementation and Results All three heuristics have been implemented by integrating them into an existing symbolic model checker for security protocols. More precisely, we modified the symbolic model checker $\mathcal{A}$SPASyA [10, 43] so that the state space is cut and searched according to the heuristics. The resulting tool namely $\mathcal{H}$-ASPASyA was tested for various protocols to analyse the gains achieved. The results show that heuristics provide a significant reduction in the size of the state space. Further the heuristics also help the searching algorithm to discover attacks rather quickly. Results also demonstrate that apart from few exceptions generally heuristic $\mathcal{H}_3$ has an edge over heuristic $\mathcal{H}_1$ and $\mathcal{H}_2$. The efficiency achieved in best cases is up till three orders of magnitude with respect to $\mathcal{A}$SPASyA.

1.4 Related Work

The heuristic search algorithms have been broadly applied in different domains including MC. Interestingly, DMC has not been the focus of the researchers working on security protocol verification. In fact, to the best of our knowledge, the only work that uses any heuristic techniques to efficiently explore the state space for security protocol verification is [46, 11]. In this section we overview some prominent DMC approaches not necessarily confined to security protocols. We also take into account any promising security protocol verification approaches that has given rise to efficient state space exploration tools.

1.4.1 Directed Model Checking

In this section, we consider some promising DMC approaches and compare the merits/demerits of the heuristics used by those works with our heuristics.

As mentioned in [41], probably the first tool that employed heuristic techniques is Approver [48]. Instead of using conventional BFS/DFS algorithms, Approver uses priority queues for searching the state space. Each element of the queue has a priority field, computed by a priority function, according to which records are ordered.

Yang and Dill, in their seminal work [36] suggested the use of heuristics to increase the bug finding capability of a model checker. Several heuristics were designed among which *target enlargement* and *tracks* were reported by the authors as the most successful heuristics. Target enlargement uses pre-images to evaluate states, where a pre-image of a state s is the set of states that can reach s in one transition. By using pre-images of error states, it is determined which states can reach the error states. However, target enlargement can only be applied during few cycles as it is memory intensive operation and it is not always possible to compute pre-images. On the other hand tracks uses approximate pre-images (based on those state variables that strongly control the behaviour of the system) and can be applied to more cycles as compared to target enlargement. Another heuristic called *guided posts* requires user defined hints for directing the search.

In [46], the model checking of concurrent systems modelled as *Calculus for Communicationg Systems*(CCS) [59] processes is coupled with heuristics to increase efficiency. A heuristic function is defined by looking at actions specified in the *property specification* formula and the CCS process to assign an estimate to a state which corresponds to the cost of reaching the goal state. The heuristic function guides the state space exploration towards interesting states and the process halts whenever a state satisfying (or violating) the property is found.

In [40], directed search algorithms are applied to verify a class of safety and liveness properties. More specifically, A* and an improved nested DFS algorithm use a heuristic exploiting the structure of promela *never claims*. A promela never claim is used to specify a property that should never be satisfied. It is in the form of Buchi automata and is executed synchronously with promela model. The global state transition graph is interpreted as a Kripke structure for which different properties are validated. Such algorithm finds shorter counter examples and performs less exploration. In contrast to [46, 40], our heuristic functions are evaluation-based and therefore do not contribute in finding shorter counter examples. Finally, the heuristics in [40] (and references therein) allow to cut the state space only in few trivial cases.

UPPAAL/DMC [53] is an extension of UPPAAL with DMC. The heuristics used in DMC are based on abstraction and are called monotonicity abstraction and automata-theoretic abstraction. The heuristic values are obtained by first extracting an abstraction of the system at hand and then calculating the error distance (i.e., the distance from a state to the error states) in this abstraction. This error distance serves as the heuristic value for the state in the concrete system. As claimed in [53], the heuristic functions defined in [53] is although computationally expensive than [46] but produces better results.

In [3], the heuristic named 'NEXT' compresses a sequence of transitions into a single meta transition. This eliminates transient states and therefore searching algorithms do less work to find the goal states. Our heuristics rely on the security formula to derive hints as against [53, 3] which rely on abstractions and process definition, respectively.

1.4.2 Verification Approaches for Security Protocols

Although general purpose tools can be used for verifying security protocols [23, 49] , researchers have also focused on designing special purpose model checkers for verifying security protocols [58, 14, 12, 33].

A brief description of some important security protocol verification techniques and tools will be presented in this section with special emphasis on OFMC [12] and Scyther [33].

Belief Logics such as BAN logic (after *Burrows Abadi Needham* logic) [25] is a verification technique in which an information exchange protocol is analysed by representing the beliefs of agents as sets of rules. This is followed by simpler and shorter proofs by exploiting these rules, which actually represent what an agent can infer from the messages it has received. One of the weaknesses of belief logics is that it does not consider the presence of compromised agents or intruder on a network and generates no counterexamples. Following the BAN logic, few other attempts such as GNY [45] and BGNY [22] have tried to address the limitations imposed by BAN logic.

Theorem proving is an inductive approach for verification that establishes the correctness of formulae by applying logical rules and inferences. When applying to security protocol verification, the required protocol is modelled as a mathematical problem with simplifications and assumptions such as perfect cryptography. The protocol is modelled as set of formulae in a logic such as modal, temporal, etc., and the property to be proved is also expressed in the same logic. The correctness of the protocol is established by proving that the property is a logical consequence of the modelled protocols. Theorem proving benefits from completion and is capable of analysing infinite state models. However, the approach requires ample amount of time to establish the correctness of the protocols. The automation of theorem proving has resulted into tools such as Isabelle [63], ETPS [6], TPS [5] and PVS [62].

Catherine Meadows presented NRL [58] as a special purpose model checker for security protocol analysis. The tool is developed in Prolog and uses a term-rewriting model of Dolev-Yao. They use a backward search strategy and the tool is mainly used for protocol verification.

Hermes [20] is a verification tool that can verify security protocols without restricting the size of messages or number of sessions. However, such verification requires the concrete model to be reduced to an abstract model with only one honest principal and one intruder. The abstraction may result in an *inconclusive* protocol which means that an attack can be found in the abstract model that is actually not present in the concrete model.

Proverif [14] is a protocol verifier based on Horn clauses. The protocol and the capabilities of the intruder are modelled in Prolog. The verifier can perform verification without constraining the number of runs of the protocol verification by introducing two approximations. Although approximations provide security guarantees they can report false attacks when there are actually no attacks. $\mathcal{H}$-ASPASyA differs from Proverif and Hermes in the verification approach; it performs bounded verification but guarantees security under a fixed scenario for a particular property. Further $\mathcal{H}$-ASPASyA can test the protocols for a variety of security properties, where as Proverif and Hermes are limited to secrecy; recently an extension in Proverif deals with authentication and correspondence also.

The AVISPA toolkit is a set of verification tools for security protocols with a common front-end. The verification process starts by first encoding the protocol in a language called High Level Protocol Specification Language (HLPSL). The HLPSL specification is then converted in to an Intermediate Format (IF) that can be handed over to each of the four back-ends provided by AVISPA. The first back-end, On-the-Fly Model Checker (OFMC) [12], employs different symbolic and constraint differentiation techniques. The result is an efficient model checker capable of performing protocol falsification as well as bounded session verification. CL-Atse [71] is another back end of AVISPA that performs both protocol falsification and verification. Furthermore, it is capable of exploiting algebraic properties of protocols. SATMC [8] first generates a propositional formula for the security problem at hand and then passes it over to a SAT-solver. Any models found are translated back as an attack. The only tool in AVISPA capable of performing unbounded verification is TA4Sp. It uses automata to represent the network and term rewriting to represent protocol and intruder knowledge. Moreover, it uses over approximation over intruder knowledge that preserves protocol correctness.

David Basin in his work presented in [11], introduced the idea of applying lazy data types to formalise protocols and the intruder. More precisely, he used a tree data structure for this purpose and used heuristics to prune and re-order the nodes. Although the idea of lazy data types is promising, the heuristics mainly are elementary. More precisely, pruning is applied to the traces with events not belonging to protocol description and re-ordering prioritises certain events in comparison to others, such as events involving the intruder are rated higher.

OFMC borrows the notion of lazy infinite state spaces from Basin's work [11] and extends it by incorporating different techniques. Mainly, the undecidability arising due to Dolev-Yao nature of the intruder is dealt with symbolic technique, and is termed as lazy intruder. The idea of lazy intruder is similar to symbolic techniques used in $\mathcal{A}$SPASyA. However, both tools differ in session generation mechanism; OFMC generates sessions symbolically, where as $\mathcal{A}$SPASyA generates initial contexts explicitly with open variables [1] in each context instantiated symbolically. Although the versions of OFMC given in [12] does not use any heuristics, the authors foresee a role of heuristics to improve the performance of OFMC. In our opinion our heuristics (mainly $\mathcal{H}_1$) can be successfully transported to OFMC and we conjecture that such experiments may produce better results.

Scyther [33] is a state-of-the-art tool that utilises the concept of heuristics for cryptographic protocol verification. The idea is to construct a pattern which is a security property to be verified , $pt = (E, \rightarrow)$, where E is a set of events and $\rightarrow$ is a relation on the events. In order to check whether a pattern can occur in an actual trace of a protocol, a set of explicit patterns is generated. If there exists any explicit pattern, it is an actual trace of the protocol and represents an attack. However, sometimes a pattern can not be realised owing to the fact that some events in the pattern are not enabled. The verification algorithm chooses an event whose terms do not have a source. The choice for the event is made on the basis of some heuristics investigated by Cremers and documented in [34]. The author reports the heuristics `privatekeys`, `constants` and `decrytpions` to be the most effective in reducing the number of states visited. `Privatekeys` heuristic gives priority to a term containing a private key as a subterm over the terms containing a public key. The heuristic `onstants` finds a ratio of constants to basic terms in the goal term and the goal with highest ratio is selected. `Decryptions` give priority to those goals that correspond to keys needed for decrypt events unless the keys are in the initial intruder knowledge.

The heuristics in [33] alike ours are evaluation-based. However, our heuristics

[1]Open variables are meant for sharing initial information such as keys and identities of the principals playing the role of the responder.

have an edge over the ones in [33] due to their pruning capability which does less exploration when there is no attack.

1.5 Structure of the Thesis

The rest of the thesis is organised as follows.

Chapter 2 This chapter discusses security protocols and a few concepts necessary to understand our work. We introduce our notations for security protocols in § 2.1, § 2.2 gives the Dolev-Yao intruder model, and § 2.3 gives types of attacks on security protocols. We enumerate and briefly comment on various formal verification techniques in § 2.4 and focus on directed model checking in § 2.5; § 2.6 gives the reader necessary understanding of the formal framework on which our work is based upon. The framework mainly comprises a process algebra for formalising security protocols (*cIP*) and a logic for defining security properties ($\mathcal{PL}$); the former is explained in § 2.6.1 and the latter in § 2.6.2.

Chapter 3 This chapter along with chapters 4 and 5 gives our core contribution. The heuristics for the formal framework (introduced in § 2.6) are defined in this chapter. We start by explaining how the state space is generated according to the semantics of *cIP* in § 3.1. The formal definition of the heuristics $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ is given respectively in § 3.2, § 3.3, and § 3.4. We briefly comment on admissibility and optimality of our heuristics in § 3.5. Finally, § 3.6 gives an intuitive approach of how heuristics have been incorporated into a searching algorithm.

Chapter 4 The algorithm and implementation details of $\mathcal{H}$-ASPASyA are reported in this chapter; § 4.1 briefly describes the architecture of $\mathcal{H}$-ASPASyA and § 4.2 gives the algorithm for $\mathcal{H}$-ASPASyA.

Chapter 5 presents our experimental results obtained by testing $\mathcal{H}$-ASPASyA with various protocols reported in Appendix A. The experiments have been conducted varying the initial input conditions for the test protocols and have been reported in § 5.1 and § 5.2. The former gives the results on state space reduction and the latter on time savings. We analyse the results in § 5.3. A few guidelines about choosing a heuristic according to the choice of the security protocol and the security property are given in § 5.4.

Chapter 6 This chapter concludes the thesis by providing a summary of our overall contribution and a few suggestions regarding how this work can be extended.

Appendix A The Appendix A provides a list of security protocols that have been taken as test cases. For each of the protocol, its encoding in cIP together with the security property to be verified is also given.

Chapter 2

Background

In this chapter, we give a few concepts necessary to understand our work. We present security protocols, an intruder model, and attacks on security protocols respectively in § 2.1, § 2.2, and § 2.3. Model checking is discussed in § 2.4. We recall the essence of informed search algorithms and directed model checking in § 2.5. Finally, § 2.6 gives the reader necessary understanding of the formal framework our work is based upon.

2.1 Security Protocols

A *security protocol* is a sequence of steps that are carried out in order to accomplish certain security goals such as confidentiality, authenticity, data integrity, non-repudiation, etc. A security protocol therefore is the abstract representation of the actions carried out by the agents participating in the communication by hiding the complexities of cryptography. Mainly, a protocol is given as a sequence of messages exchanged between communicating parties where cryptographic primitives are represented by appropriate notations. We introduce the notations for symmetric key cryptography by means of a simple example.

$$\begin{aligned} &1.\ \ A \rightarrow S:\ \{T_a, B, k_{ab}\}_{k_{as}} \\ &2.\ \ S \rightarrow B:\ \{T_s, A, k_{ab}\}_{k_{bs}} \end{aligned} \tag{2.1}$$

We use the narration given in 2.1 (wide-mouthed frog protocol), to illustrate the key ingredients in any protocol, viz.; *participants*, *messages*, and cryptographic *notations*. Participants are represented by capital letters, such as A, B etc. Thus, the notation $A \rightarrow S$ is used to represent a message initiated by participant identified as A, whose intended recipient is S. Messages exchanged during the communication can be simple or composed, for instance in 2.1, $\{T_a, B, k_{ab}\}_{k_{as}}$ is an example of a composed message sent over a network. The message is obtained by composing three components, a timestamp generated by A (T_a), the identity of the participant with which A wants to communicate (B), and a fresh session key generated by A to be shared with B for secure communication (k_{ab}). Message $\{T_a, B, k_{ab}\}_{k_{as}}$ is encrypted by the secret key k_{as} shared between the server and A. In the second step, the server forwards the session key along with the identity of the initiator and his own timestamp to B, encrypted with k_{bs}.

We use the Needham Schroeder (NS) public key protocol to introduce notations for public key cryptography. The NS protocol consists of the following steps

$$\begin{aligned} &1.\ \ A \rightarrow B:\ \ \{na, A\}_{B^+} \\ &2.\ \ B \rightarrow A:\ \ \{na, nb\}_{A^+} \\ &3.\ \ A \rightarrow B:\ \ \{nb\}_{B^+} \end{aligned}$$

where, in step 1 the initiator A sends to B a nonce na and her identity encrypted with B's public key B^+; in step 2, B responds to the nonce challenge by sending to A a fresh nonce nb and na encrypted with A^+, the public key of A; A concludes the protocol by sending back to B the nonce nb encrypted with B's public key.

2.2 An Intruder Model

The design of *secure* cryptographic protocols despite of their apparent simplicity is hard. Protocols usually operate on networks subject to the presence of malign agents called *intruders* (also called attackers or spies) who can engineer a security breach by manipulating any loopholes left by the protocol designers. An *intruder model* formalises the capabilities of an intruder and describes what he is entitled to do. An intruder is usually modelled such that he has unbound memory and computational power. A number of intruder models have been proposed to date among which the Dolev-Yao intruder model [37] is the most powerful and commonly used one.

A Dolev-Yao intruder controls the communication of a network and can store, destroy, and modify any messages communicated on the network. Further he can generate a possibly infinite set of messages from his knowledge to match any of the inputs expected by a participant of a protocol. He can achieve this through pairing, encryption, and decryption of the messages, provided he has appropriate keys (perfect cryptography assumption). Thus he can apply pairing to generate a message such as (m,n) if he can retrieve m and n from his knowledge. He can generate a messages like $\{m\}_k$ provided he has the message m and the key k in his knowledge. Likewise, he can decrypt a message $\{n\}_k$ only when he has appropriate decrypting key. An attack on the Needham Schroeder protocol given in § 3.2.2 demonstrates how Dolev-Yao intruder can manage security breach using his powers.

2.3 Types of Attacks

A protocol can be vulnerable to many different kinds of threats when deployed on a network. We list here three major kinds of threats a protocol can face.

A *replay* attack is a kind of passive attack where an attacker listens the communication and records it. He then later uses the information to make false claims such as false identification or authentication.

A *man in the middle* attack is the one in which the communicating parties are deceived by the attacker in such a way so as to make them believe they are communicating with each other. In reality the whole communication is controlled by the attacker. An attacker establishes two sessions, one with the sender and other with the receiver. He then relays the information between the sender and receiver, claiming to be the other party in each case.

A *reflection* attack is an attack in which information from multiple sessions is used by the attacker to complete any one of the sessions. He then simply abandons the other sessions.

2.4 Model Checking

As stated in § 2.1, it is hard to design and implement correct protocols. Although short and apparently simple, their design can contain inherent flaws that can lead to severe security risks. A carefully designed protocol can later turn out to be faulty due to the fact that it is difficult as a human to see all possible assumptions that are required for a protocol to work. Furthermore, an intruder is highly non-deterministic in nature and to foresee what he can manoeuvre through his tactics is not an easy task. Formal verification of security protocols is therefore important and helps in two ways: (i) formal representation of the protocols clarifies the assumptions and ambiguities in informal representation for the protocol and (*ii*) verifying the protocol either proves its correctness or reveals flaws in it. State space exploration techniques such as MC aim at representing every possible behaviour of the protocol. The method generates the state space associated with the protocol and try to establish if certain condition (property) holds at each state. However, analysing a protocol for arbitrary number of participants, multiple protocol runs, and unbounded message size, may generate infinite state spaces. Consequently, state space exploration techniques and tools based on them use some form of restrictions so as to make state space finite.

Model checking is quite a popular technique for security protocol verification. It is a completely automatic process that is decidable and generates counterexamples that help the verifier to track down the error state. A number of general purpose model checkers such as SMV [24], Spin [49], Mocha [2], etc. have been used for the verification of security protocols. Furthermore, special purpose security protocol model checkers have also been introduced, such as AVISPA tool kit [7]. Other tools such as Athena [68] and Scyther [34] combine the concepts of model checking and theorem proving.

Model checking expects a protocol to be modelled as a transition system with finitely many states. The model is then tested against a property that is usually expressed in some form of logic such as temporal or modal. The verification phase reduces to a state space search problem that attempts to check whether property holds in the given transition system or not.

The potential and widely researched problem in model checking is the exponential growth of the state space (called state space explosion) as the number of processes increase, generating huge possibly infinite state spaces. It is therefore mandatory to consider few assumptions so as to make process decidable. The assumptions may require a bound on the number of participants, sessions, or on the size of message. Whereas these restrictions make the verification process decidable, the negative impact of these restrictions is that correctness of the protocol remains in question. This amounts to say that if a flaw is detected in the protocol, the protocol is considered to be incorrect; however, absence of the flaws does not guarantee the correctness.

One of the recent approaches for efficient exploration of the state space so that a counterexample is generated before system resources exhaust is directed model checking. We discuss DMC in the next section.

2.5 Directed Model Checking

As stated in § 1.1 state space explosion problem can be dealt with approaches like abstraction [31], symmetry [42, 30], partial order reduction [32, 44], or symbolic approaches [4, 18, 19, 43]. However, even with the use of such approaches the search space can grow enormously. Conventional MC uses blind search algorithms (such as *depth first search* (DFS) or *breadth first search* (BFS)) to explore the state space in a pre-defined order so as to reach the error states. This may result into a situation where the system runs out of its resources before any conclusion is established. Lately, many researchers have investigated on incorporating intelligent search strategies into model checking. The combination of intelligent search strategies and model checking has lead to directed model checking. The term has been coined in [65] adn further promoted and developed in [54], and characterises a model checking approach that adopts informed search algorithms instead of blind search algorithms to efficiently search the state space.

2.5.1 Informed Search Algorithms

Informed search (or heuristics search) algorithms aim to explore/generate state spaces efficiently by applying a simplification, a rule of thumb, or an educated guess. The idea is to decide, at each branching state, which path to follow so to reach the goal node rather quickly. The decision is usually taken with the help of a *heuristic function* that contains domain specific hints (called heuristics). The hints are in addition to the actual problem (the goal) and help in assigning heuristic values to the nodes as they are encountered during exploration. The heuristic values, also called *estimates*, influence the order in which the nodes are to be explored as against blind search algorithms where order of exploration is already fixed. Such algorithms do not guarantee optimal solutions as the estimates are not always accurate. The trade off here is for efficiency which is required when complete state space exploration is not feasible or requires extensive time to complete the search. The examples include variants of *best first search* (BFS) such as greedy best first search or A and A* algorithms. Such algorithms evaluate each node according o a heuristic function and explore the nodes starting from those with lowest heuristic values. For instance, greedy best first search selects nodes that are closest to goal; if the heuristic value of the successor s of a node n is higher that the value of n, the search continues from s (otherwise, s is en-queue according to its heuristic value).

Algorithm 1 given below gives a basic BFS algorithm.

Algorithm 1 `BestFirstSearch`

```
1: open ← $n_0$.
2: while open ≠ ∅ do
3:     Select best n from open
4:     if n = goal then
5:         return path (by backtracing path to n).
6:     else
7:         Expand n.
8:         Using heuristic function evaluate each successor node.
9:         Add successors to open.
10:    end if
11:    Go to step 2
12: end while
```

In the first step of Algorithm 1, a list `open` is created with initial node n_0. In the next step, the node `n` with best heuristic value is selected. If `n` is a goal node then algorithm terminates and return the path to n; otherwise, the successors of n are generated and evaluated according to the heuristic function. Each successor node is then added to the list open. The process is repeated until a goal node is found or open is empty.

Algorithm A uses an evaluation function $f(n) = g(n) + h(n)$, where $h(n)$ is the heuristic estimate for the node n and $g(n)$ is the cost to reach the node n from an initial node n_0. Algorithm A is not optimal, however a variant of it called A* generates optimal solutions by using an *admissible heuristic function* [54].

2.5.2 Basics of Heuristics

We recall here the basic concepts on heuristics by means of a simple example. A deeper presentation can be found in [66].

The n-puzzle (also known as the sliding-block or tile-puzzle) is a well-known puzzle in which the goal is to move square tiles by sliding them horizontally or vertically in one empty tile. For $n = 8$ the goal configuration is depicted in Figure 2.1; a possible initial configuration is in Figure 2.2. The problem of finding the shortest path leading to the goal configuration is NP-hard.

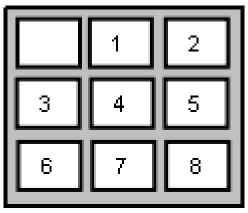


Figure 2.1: The 8-puzzle goal configuration

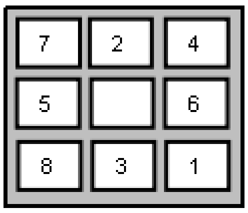


Figure 2.2: A possible start configuration

A very simple heuristic (cf. [66]) for 8-puzzle can be given by

$$h_1 = \text{number of misplaced tiles.}$$

For each configuration, h_1 counts the number of misplaced tiles with respect to the goal configuration. For instance, h_1 weights 8 the configuration in Figure 2.2 since all the tiles are misplaced.

Another heuristic (cf. [66]) for 8-puzzle is the one that exploits the so called *Manhattan distance*.

$$h_2 = \text{sum of the Manhattan distances of non-empty tiles from their target positions.}$$

So the configuration in Figure 2.2 is weighted 18 by h_2.

2.5.2.1 Classification of Heuristic Functions

Heuristic functions can be categorised on the basis of the heuristic values they assign to nodes. In [54], the author reports two categories of heuristic functions, namely estimation functions and evaluation functions. An *estimation function* is a heuristic function that assigns a weight to the node by estimating its *distance* from the goal node. On the other hand an *evaluation function* assigns the weight to a node according to the *possibility* that the node will lead to a goal node.

2.5.2.2 Types of Heuristics

Bloem et al, [15] classify heuristics as *system-dependent* and *property dependent*. The former refers to the hints that are obtained by looking at the design of the program to avoid bottlenecks of computations and the latter to the ones that guide the search towards property violation (or preserving) states. Property dependent heuristics are further classified as *property-specific* heuristics and *structural* heuristics. Property specific heuristics derive the hints from the specification of property itself, while structural heuristics rely on the structure of the program to derive the hints.

2.6 Formal Framework

We adopt the formal verification framework introduced in [43] consisting of the *cIP* (after *cryptographic Interaction Pattern*) process calculus and the $\mathcal{PL}$ logic (after *Protocol Logic*) to respectively represent security protocols and properties. We refer the reader to [43] for a precise and detailed presentation of the framework.

2.6.1 A Process Calculus for Security Protocols

cIP is a process calculus to formally represent a security protocol. A principal is represented as a *cIP* process, where a *cIP* process can have an identity, a set of open variables, and actions it can perform. In *cIP*, a process is given as

$$\begin{array}{lcl} P & ::= & PN = (X)[E] \\ E & ::= & 0 \mid \alpha.E \mid E||E \mid E+E \\ \alpha & ::= & in(M) \mid out(M) \end{array}$$

where *PN* is the identity of the process, *X* is the set of *open* variables of the process; open variables are meant for sharing initial conditions, such as symmetric keys or identity of the responder, and *E* is the behavioural expression of the process. A behavioural expression of a process mainly represents actions a process can perform. A behavioural expression of a process can be an inaction (0), an action performed by the process followed by behavioural expression $(\alpha.E)$, a parallel composition of expressions $(E||E)$, and non-deterministic composition of behavioural expressions $(E+E)$. A process can perform two types of actions, *out(M)* and *in(M)*, that correspond to *sending* and *receiving* of message *M* on a network.

A principal *instance* can join a protocol *session* by means of open variables. An *instance* of a principal is obtained by properly indexing his identity and messages with a natural number. A *cIP session* is a finite set of instances of principals and is represented by

$$\{(X_{i_1})[E_{i_1}],\ldots,(X_{i_n})[E_{i_n}]\}$$

2.6.1.1 Formalising the Needham Schroeder Protocol

We illustrate the syntax of *cIP* by formalising the NS protocol:

$$\begin{array}{llll} A:(r)[& out(\{na,A\}_{r^+}). & B:()[& in(\{?x,?y\}_{B^-}). \\ & in(\{na,?z\}_{A^-}). & & out(\{x,nb\}_{y^+}). \\ & out(\{z\}_{r^-}) \;\;] & & in(\{nb\}_{B^-}) \;\;] \end{array} \tag{2.2}$$

The principal A (resp. B) in (2.2) represents the initiator (resp. the responder) of the NS protocol. The open variable r is meant to be bound to the identity of the responder. The principal A first executes the output action and then waits for a message expected to match the pattern specified in the *in* action. More precisely, A will receive any pair encrypted with her public key whose first component is the nonce *na*; upon a successful match, the second component of the pair will be assigned to the variable z. For instance, the $\{na,M\}_{A^+}$ matches $\{na,?z\}_{A^-}$ for any M and would assign M to z.

The instance of the NS initiator obtained by indexing the principal A in (2.2) with 2 is

$$A_2:(r_2)[out(\{na_2,A_2\}_{r_2^+}).in(\{na_2,?z_2\}_{A_2^-}).out(\{z_2\}_{r_2^+})] \tag{2.3}$$

and a possible session for the NS protocol in which A_2 participated together with B_1 is

$$\begin{array}{c} \{()[out(\{na_2,A_2\}_{B_1^+}).in(\{na_2,?z_2\}_{A_2^-}).out(\{z_2\}_{B_1^+})] \\ ()[in(\{?x_1,?y_1\}_{B_1^-}).out(\{x_1,nb_1\}_{y_1^+}).in(\{nb_1\}_{B_1^-})]\} \end{array} \tag{2.4}$$

2.6.1.2 Features of *cIP*

cIP provides a number of unique features that allow one to test security protocols under different conditions. The open variables in *cIP* are means to specify initial conditions; in particular they are used by the principals joining a session to acquire necessary information such as shared keys. Thus open variables together with join formulae (cf., § 2.6.2) allow the verifier to specify interesting verification scenarios. Furthermore, one can test the protocols with varying intruder knowledge and number of principal instances participating in a session.

As mentioned earlier, the design of security protocol is error prone and their analysis is hard due to the generality of the intruder model considered for verification. One capability of the intruder model is to gather information from multiple sessions and exploit it to devise an attack. *cIP* is capable of identifying such attacks as it supports *multi-session* analysis. Principals participating in a session are indexed and $\mathcal{PL}$ uses quantification over indexed principals to specify properties pertinent to particular instances of the principals. This helps in bringing out any attacks engineered by the intruder by manipulating information from various sessions.

Another issue in security protocol analysis is *infinite branching*. The problem arises as the intruder can generate infinitely many messages from his available knowledge using operations such as pairing, encryption, and decryption. *cIP* addresses the infinite branching problem with the help of symbolic semantics. The idea is to delay the actual instantiation of the variables, which amounts to say that a variable is not instantiated immediately but is rather kept symbolic. Intuitively, a symbolic mapping replaces a variable x with a symbolic variable $x[\kappa]$ meaning that x can be assigned any message derivable from κ. The actual instantiation will possibly assign (infinitely) many messages to the variable, derivable from the associated intruder knowledge. Thus actual instantiation of x is delayed till a concrete message is needed.

2.6.2 Logic for Security Property Specification

We adopt the definition of $\mathcal{PL}$ formulae given in [43]:

$$\phi, \psi \ ::= \ x_i = m \ \mid \ \kappa \triangleright m \ \mid \ \forall i : A.\psi \ \mid \ \exists i : A.\psi \ \mid \ \neg\psi \ \mid \ \psi \wedge \phi \ \mid \ \psi \vee \phi$$

where x_i are indexed variables.

The atomic formulae $x_i = m$ and $\kappa \triangleright m$ hold respectively when the variable x_i is assigned the message m and when κ (representing the intruder's knowledge) can derive m. A formula can universally and existentially quantify over indexes i because $\mathcal{PL}$ predicates over the *instances* of the principals concurrently executed. A $\mathcal{PL}$ formula can be a composed $\mathcal{PL}$ formula by using operators such as $\neg$, $\wedge$, and $\vee$.

As an example of $\mathcal{PL}$ formula consider the formula ψ_{NS} predicating on (instances of) the NS protocol:

$$\forall i : A.\ \exists j : B\ (x_j = na_i\ \wedge\ z_i = nb_j).$$

The formula ψ_{NS} states that for all instances of A there should be an instance of B that has received the nonce na_i sent by A_i and the nonce nb_j is received by the instance A_i.

Another interesting feature of the framework is the use of *join* formulae, which permits to specify constraints on the way principals can be connected to each other. In this way, one can specify only those scenarios one is interested to verify. For instance, consider following *join* formula for NS protocol:

$$(\exists j : A.\ true)\ \wedge\ (\exists i : B.\ true)$$

which specifies that we are interested only in those runs of verification in which there is at least an instance of A and an instance of B.

Chapter 3

Heuristics for the Verification of Security Protocols

This chapter introduces three heuristics (called $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$) for efficientlyy searching the state space generated for model checking security protocols. The heuristics are *property-specific* (see § 2.5) and are defined on the security formula expressed in $\mathcal{PL}$. They are efficient as they (i) can guide the searching algorithm towards promising regions of the state space, and (ii) can prune those parts of the state space where there is no chance of attack under a given security formula.

The heuristic $\mathcal{H}_1$ is defined in terms of two mutually recursive functions $\mathcal{H}_s$ and $\mathcal{H}_t$ which assign weights to states and transitions, respectively. Heuristic function $\mathcal{H}_2$ assigns weights to states by exploiting $\mathcal{PL}$ formulae and the way open variables are assigned values in the states. Heuristic $\mathcal{H}_3$ is a composite heuristic as it combines $\mathcal{H}_1$ and $\mathcal{H}_2$.

The state space is obtained according to the semantics of *cIP* defined in [43]. The state space is informally presented with the help of the NS example in § 3.1. The heuristics $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ are given respectively in § 3.2, § 3.3 and § 3.4. An intuitive idea of how heuristics are incorporated in the searching algorithm is given in § 3.6.

3.1 The State Space

A state consists of a tuple $\langle \mathcal{C}, \chi, \kappa \rangle$ where

- $\mathcal{C}$ is a context containing principal instances which joined the session,
- χ is a mapping of variables to messages, and
- κ is a set of messages representing the intruder knowledge.

A transition from one state to another can be the result of *out* and *in* actions performed by principal instances or of *join* operations; join transitions may instantiate open variables by assigning them with the identity of some principal. Initially, $\mathcal{C}$ is empty and therefore only join transitions are possible. When $\mathcal{C}$ contains an instance ready to send a message, an *out* transition can be fired so that the sent message is added to κ. If $\mathcal{C}$ contains a principal ready to receive a message, the intruder tries to derive from the messages in κ a message that matches the pattern specified in the input action; if such a message is found, χ is updated to record the assignments to the variables occurring in the input action.

For instance, a few possible transitions for the NS protocol (described in § 2.6.1.1) are

$$s_0 \xrightarrow{join} s_1 \xrightarrow{join} s_2 \xrightarrow{out} s_3 \xrightarrow{in} s_4$$

where $s_0 = \langle\ \emptyset, \emptyset, \kappa_0\ \rangle$ with $\kappa_0 = \{I, I^+, I^-\}$, namely initially no principal instance joined the context, there is no assignment to variables, and the intruder only knows its identity and public/private keys.

The join transition from s_0 to s_1 adds a principal instance B_2 to the context yielding

$$\begin{aligned}
s_1 =\langle\ & \{()[in(\{?x_2, ?y_2\}_{B_2^-}).out(\{x_2, nb_2\}_{y_2^+}).in(\{nb_2\}_{B_2^-})]\}, \\
& \emptyset, \\
& \kappa_1 = \kappa_0 \cup \{B_2, B_2{}^+\}\ \rangle
\end{aligned}$$

that is, the intruder now knows B_2's identity and (by default) its public key. Similarly, the transition from s_1 to s_2 adds the principal instance A_1 to the context and therefore

$$\begin{aligned}
s_2 = \langle\ & \{()[in(\{?x_2, ?y_2\}_{B_2^-}).out(\{x_2, nb_2\}_{y_2^+}).in(\{nb_2\}_{B_2^-})], \\
& ()[out(\{na_1, A_1\}_{B_2^+}).in(\{na_1, ?z_1\}_{A_1^-}).out(\{z_1\}_{B_2^+})]\}, \\
& \{r_1 \mapsto B_2\}, \\
& \kappa_2 = \kappa_1 \cup \{A_1, A_1{}^+\}\ \rangle
\end{aligned}$$

Notice that the open variable r_1 is now mapped to B_2.

The transition from s_2 to s_3 is due to an *out* action performed by A_1

$$\begin{aligned}
s_3 = \langle\ & \{()[in(\{?x_2, ?y_2\}_{B_2^-}).out(\{x_2, nb_2\}_{y_2^+}).in(\{nb_2\}_{B_2^-})], \\
& ()[in(\{na_1, ?z_1\}_{A_1^-}).out(\{z_1\}_{B_2^+})]\}, \\
& \{r_1 \mapsto B_2\}, \\
& \kappa_3 = \kappa_2 \cup \{na_1, A_1\}_{B_2^+}\ \rangle
\end{aligned}$$

the prefix of A_1 is consumed and the message is added to the intruder's knowledge.

Finally, the transition from s_3 to s_4 is due to an *in* transition for the input prefix of B_2. The message $\{na_1, A_1\}_{B_2^+}$ added to the intruder's knowledge in the previous transition matches the pattern $\{?x_2, ?y_2\}_{B_2^-}$ specified by B_2, therefore the x_2 and y_2 are assigned na_1 and A_1 respectively. Hence,

$$\begin{aligned} s_4 = \langle\ & \{()[out\{na_1, nb_2\}_{A_1^+}).in(\{nb_2\}_{B_2^-})], \\ & ()[in(\{na_1, ?z_1\}_{A_1^-}).out(\{z_1\}_{B_2^+})]\}, \\ & \{r_1 \mapsto B_2, x_2 \mapsto na_1, y_2 \mapsto A_1\}, \\ & \kappa_3\ \rangle \end{aligned}$$

In this framework, join transitions can be safely anticipated before any other transition (Observation 10.1.3 in [70], page 174).

3.2 A First Heuristic $\mathcal{H}_1$

The first heuristic $\mathcal{H}_1$ relies on two mutually recursive functions $\mathcal{H}_s$ and $\mathcal{H}_t$ (given in Definition 2 and Definition 3 respectively) and exploits the security property expressed in $\mathcal{PL}$ to assign weights to states and transitions. The heuristic $\mathcal{H}_1$ exploits the "quantifiers" of the security formula and the "context" (signifying the principal instances which joined the session) of a state to decide heuristic values for states and transitions. Intuitively, a formula universally quantified over principal instances can be falsified when at least one instance of such principal has joined the session. On the other hand, a formula with existential quantification over principal instances is negated in the contexts that do not contain any such instance.

For simplicity and without loss of generality, we define the heuristic on *Prenex Normal Form* (PNF) formulae defined below.

Definition 1 (Prenex Normal Form)**.** *A $\mathcal{PL}$ formula is in* prenex normal form *if it is of the form*

$$\mathtt{Q}_1 i_1 : A_1. \cdots .\mathtt{Q}_n i_n : A_n.\phi$$

where ϕ is a quantifier-free formula, for $1 \leq j \leq n$, $\mathtt{Q}_j \in \{\forall, \exists\}$, each i_j is an index variable, and A_j is a principal name.

Basically, a PNF formula is a formula where all the quantifiers are "at top level". Notice that, in Definition 1, it can be $n = 0$ which amounts to say that a quantifier free formula is already in PNF.

Theorem 3.2.1. *Any $\mathcal{PL}$ formula can be transformed into a logically equivalent PNF formula.*

Proof. Let the function $pnf : \mathcal{PL} \rightarrow \mathcal{PL}$ be defined as follows:

$$pnf(\psi) = \begin{cases} \psi & \psi \text{ is a quantifier-free formula.} \\ \mathtt{Q}\ i : A.pnf(\psi') & \psi \equiv \mathtt{Q}\ i : A.\psi' \\ \mathtt{Q}\ i' : A.pnf(\psi_1'[i'/i] \wedge \psi_2) & i' \text{ fresh}, \psi \equiv \psi_1 \wedge \psi_2 \text{ and } pnf(\psi_1) \equiv \mathtt{Q}\ i : A.\psi_1' \\ \mathtt{Q}\ i' : A.pnf(\psi_1'[i'/i] \vee \psi_2) & i' \text{ fresh}, \psi \equiv \psi_1 \vee \psi_2 \text{ and } pnf(\psi_1) \equiv \mathtt{Q}\ i : A.\psi_1' \\ \overline{\mathtt{Q}}\ i : A.pnf(\neg\psi'') & \psi \equiv \neg\psi' \text{ and } pnf(\psi') \equiv \mathtt{Q}\ i : A.\psi'' \end{cases}$$

where $\equiv$ defines the syntactic equality between formulae and $\overline{\mathtt{Q}}$ represents negation of $\mathtt{Q}$. The proof of theorem 3.2.1 follows from the properties of *pnf* given by Lemmas 3.2.2 and 3.2.3 below. □

Lemma 3.2.2. *For any $\mathcal{PL}$ formula ψ, $pnf(\psi)$ is in PNF.*

Proof. We proceed by induction on the structure of ψ.

If ψ is a quantifier free formula then it is in PNF and, by definition of *pnf*, $pnf(\psi) = \psi$.

The inductive case is proved by case analysis.

- Assume ψ is $\mathtt{Q}\ i : A.\psi'$, then by definition of *pnf*, $pnf(\psi) = \mathtt{Q}\ i : A.pnf(\psi')$. By inductive hypothesis $pnf(\psi')$ is in PNF and therefore $pnf(\psi)$ is in PNF.

- If $\psi = \psi_1 \wedge \psi_2$ then, assuming $pnf(\psi_1) = \mathtt{Q}\ i : A.\psi_1'$, by definition of *pnf*

$$pnf(\psi) = \mathtt{Q}\ i' : A.pnf(\psi_1'[i'/i] \wedge \psi_2)$$

  For fresh index i' not occurring in ψ_2. By the inductive hypothesis $pnf(\psi_1'[i'/i] \wedge \psi_2)$ is in PNF, therefore $pnf(\psi)$ is in PNF.

- The case $\psi = \psi_1 \vee \psi_2$ is analogous.

- If $\psi = \neg\psi'$ then, assuming $pnf(\psi') = \mathtt{Q}\ i : A.\psi''$, by definition of *pnf*, $pnf(\psi) = \overline{\mathtt{Q}\ i} : A.pnf(\neg\psi'')$. By inductive hypothesis $pnf(\neg\psi'')$ is in PNF, therefore $pnf(\psi)$ is in PNF.

□

Lemma 3.2.3. *$pnf(\psi) \Leftrightarrow \psi$.*

Proof. We proceed by induction on the structure of ψ.

If ψ is a quantifier free formula then $pnf(\psi) = \psi$ and therefore $pnf(\psi) \Leftrightarrow \psi$.

Again the proof for the inductive case is given by case analysis.

- Assume ψ is $\mathtt{Q}\ i : A.\psi'$, then by definition of *pnf*, $pnf(\psi) = \mathtt{Q}\ i : A.pnf(\psi')$. By inductive hypothesis $pnf(\psi') \Leftrightarrow \psi'$ hence $pnf(\psi) \equiv \mathtt{Q}\ i : A.\psi'$ and therefore $pnf(\psi) \Leftrightarrow \psi$.

- If $\psi = \psi_1 \wedge \psi_2$ then, assuming $pnf(\psi_1) = \mathtt{Q}\ i : A.\psi_1'$, by definition of *pnf*

$$pnf(\psi) = \mathtt{Q}\ i' : A.pnf(\psi_1'[i'/i] \wedge \psi_2)$$

  For i' fresh (namely, i' does not occur in ψ_2). By the inductive hypothesis $pnf(\psi_1) \Leftrightarrow \psi_1$ hence

$$\psi \Leftrightarrow pnf(\psi_1) \wedge \psi_2 = (\mathtt{Q}\ i : A.\psi_1') \wedge \psi_2$$

It is trivial to prove that for any $\mathcal{PL}$ formula $(\mathtt{Q}\, i : A.\psi) \wedge \phi \Leftrightarrow \mathtt{Q}\, i : A.(\psi \wedge \phi)$ and therefore $pnf(\psi) \Leftrightarrow \psi$.

- The proof for $\psi = \psi_1 \vee \psi_2$ is similar.
- If $\psi = \neg\psi'$, assume $pnf(\psi') = \mathtt{Q}\, i : A.\psi''$. By definition, $pnf(\psi) = \overline{\mathtt{Q}\, i} : A.pnf(\neg\psi'')$. By inductive hypothesis $pnf(\psi') \Leftrightarrow \psi'$ and therefore $\psi \Leftrightarrow \neg pnf(\psi')$, hence $\psi \Leftrightarrow \neg(\mathtt{Q}\, i : A.\psi'') \Leftrightarrow \overline{\mathtt{Q}\, i} : A.\neg\psi''$ and therefore $pnf(\psi) \Leftrightarrow \psi$.

□

The heuristic function $\mathcal{H}_s$ is given in Definition 2 and depends on the function $\mathcal{H}_t$ given in Definition 3 below.

Definition 2 (Weighting states)**.** *Given a state s and a formula* ϕ*, the* state weighting function *is given by*

$$\mathcal{H}_s(s,\phi) = \begin{cases} \max\limits_{t \in s\Upsilon} \mathcal{H}_t(t,\phi), & s\Upsilon \neq \emptyset \\ -\infty, & \phi \equiv \forall i : A.\ \phi' \ \wedge\ s\Upsilon = \emptyset \ \wedge\ s \cap [A] = \emptyset \\ 0, & \textit{otherwise} \end{cases}$$

where $s\Upsilon$ *is the set of join transitions departing from* s*,* $[X]$ *be the set of all instances of a principal* X*, and assuming* $s = \langle \mathcal{C}, \chi, \kappa \rangle$*,* $s \cap [A]$ *stands for* $\kappa \cap [A]$.

The function $\mathcal{H}_s$ takes a state, say $s = \langle \mathcal{C}, \chi, \kappa \rangle$, and a formula ϕ as input and returns the maximum among the weights computed by $\mathcal{H}_t$ on the join transitions departing from s for ϕ. The weight $-\infty$ is returned if

- ϕ is a universal quantification on a principal instance A ($\forall i : A.\ \phi'$),
- s does not have outgoing join transitions ($s\Upsilon = \emptyset$), and
- there is no instance of A in the context ($s \cap [A] = \emptyset$).

The heuristic $\mathcal{H}_s$ has been designed considering that a formula universally quantified on instances of A is falsified in those states where there is at least one instance of A. Therefore a context without an instance of the quantified principal, has no chance of falsifying the formula. In fact, the condition $s\Upsilon = \emptyset$ ensures that no principal instance can later join the context. As a result, there is no possibility of falsifying the property in all paths emerging from this state which can therefore be pruned.

The heuristic that assigns weights to transitions is given in Definition 3.

Definition 3 (Weighting transitions)**.** *Given a state s and a transition t from s to $s' = \langle C', \chi', \kappa' \rangle$ in $s\Upsilon$, the* weighting transitions *function $\mathcal{H}_t$ is*

$$\mathcal{H}_t(t,\phi) = \begin{cases} 1 + \mathcal{H}_s(s',\phi'), & \phi \equiv \forall i : A.\, \phi' \wedge \kappa' \cap [A] \neq \emptyset, \\ 1 + \mathcal{H}_s(s',\phi), & \phi \equiv \exists i : A.\, \phi' \wedge \kappa' \cap [A] = \emptyset, \\ \mathcal{H}_s(s',\phi'), & \phi \equiv \exists i : A.\, \phi' \wedge \kappa' \cap [A] \neq \emptyset, \\ \mathcal{H}_s(s',\phi), & \phi \equiv \forall i : A.\, \phi' \wedge \kappa' \cap [A] = \emptyset, \\ 0 & \mathit{otherwise}. \end{cases}$$

The function $\mathcal{H}_t$ takes as input a transition t and invokes $\mathcal{H}_s$ to compute the weight of t depending on the structure of the formula ϕ. As specified in Definition 3, the value of the weight of the arrival state is incremented if either of the two following mutually exclusive conditions hold:

- ϕ universally quantifies on instances of A ($\forall i : A.\, \phi'$) for which some instances have already joined the context ($\kappa' \cap [A] \neq \emptyset$);
- ϕ existentially quantifies on an instance of A ($\exists i : A.\, \phi'$) which is not present in the context ($\kappa' \cap [A] = \emptyset$).

Instead, the heuristic $\mathcal{H}_t$ assigns the weight of its arrival state if either of the following mutually exclusive[1] conditions hold:

- ϕ existentially quantifies on an instance of A ($\exists i : A.\ \phi'$) which is present in the context ($\kappa' \cap [A] \neq \emptyset$);
- ϕ universally quantifies on instances of A ($\forall i : A.\ \phi'$) and the context does not contain any such instance ($\kappa' \cap [A] = \emptyset$).

Again the intuition behind $\mathcal{H}_t$ is based on quantifiers. The formula ϕ that universally (resp. existentially) quantifies on instances of A can be falsified only if such instances will (resp. not) be added to the context. Therefore all transitions that (resp. do not) add an instance of A get a higher value.

It is important to mention that in the first and third cases of Definition 3, the recursive call to $\mathcal{H}_s$ takes as input ϕ', the sub formula of ϕ in the scope of the quantifier. This is due to the fact that once an instance of the quantified principal has been added we are not interested in more instances and therefore consume the quantifier. The heuristic $\mathcal{H}_t$ returns 0 when ϕ is a quantifier free formula. In fact, due to the absence of quantifiers we cannot assess how promising is t to find an attack for ϕ.

Finally, we remark that $\mathcal{H}_s$ and $\mathcal{H}_t$ terminates on a finite state space because the sub-graph consisting of the join transitions forms a tree by construction. Therefore, the recursive invocations from $\mathcal{H}_t$ to $\mathcal{H}_s$ will eventually be resolved by the last two cases of $\mathcal{H}_s$ in Definition 2.

3.2.1 Correctness of $\mathcal{H}_1$

Theorem 3.2.4 states the correctness of the heuristic $\mathcal{H}_1$, namely pruned parts of the state space do not contain an attack. We first present the models for $\mathcal{PL}$ formulae borrowed from [43].

[1]Note that all the conditions of the definition of $\mathcal{H}_t$ are mutually exclusive.

Model for $\mathcal{PL}$ formulae

Definition 4 (Model for $\mathcal{PL}$ formulae)**.** *Let χ be a substitution from indexed variables to indexed messages, κ a knowledge and ϕ a closed formula of $\mathcal{PL}$. Then $\langle \kappa, \chi \rangle$* is a model for ϕ *if $\kappa \models_\chi \phi$ can be proved by the following rules (where n stands for an instance index):*

$$\frac{xa_n\chi = m\chi}{\kappa \models_\chi xa_n = m}\,(=) \qquad \frac{\kappa \triangleright m\chi}{\kappa \models_\chi \kappa \triangleright m}\,(\triangleright)$$

$$\frac{\textit{exists } n \textit{ s.t. } A_n \in \kappa \quad \kappa \models_\chi \phi[n/i]}{\kappa \models_\chi \exists i : A.\ \phi}\,(\exists)$$

$$\frac{\textit{forall } n \textit{ s.t. } A_n \in \kappa \quad \kappa \models_\chi \phi[n/i]}{\kappa \models_\chi \forall i : A.\ \phi}\,(\forall)$$

$$\frac{\kappa \models_\chi \phi \quad \kappa \models_\chi \psi}{\kappa \models_\chi \phi \wedge \psi}\,(\wedge) \qquad \frac{\kappa \models_\chi \phi}{\kappa \models_\chi \phi \vee \psi}\,(\vee 1)$$

$$\frac{\kappa \models_\chi \psi}{\kappa \models_\chi \phi \vee \psi}\,(\vee 2) \qquad \frac{\kappa \not\models_\chi \phi}{\kappa \models_\chi \neg\phi}\,(\neg)$$

Theorem 3.2.4. *If $\mathcal{H}_s(s, \phi) = -\infty$ then for any state s' = $\langle C', \chi', \kappa' \rangle$ reachable from s= $\langle C, \chi, \kappa \rangle, \kappa' \not\models_{\chi'} \neg\phi$*

Proof. If $\kappa' \models_{\chi'} \neg\phi$, by Definition of $\models$, there is $A_n \in \kappa'$(because $\phi \equiv \forall i : A.\ \phi'$). However by hypothesis $s \cap [A] = \emptyset$ and $s\Upsilon = \emptyset$ hence $s' \cap [A] = \emptyset$ and therefore s' does not satisfy $\neg\phi$. □

3.2.2 Evaluation of $\mathcal{H}_1$

In this section we illustrate the efficiency of our approach with the help of examples. In the first example $\mathcal{H}_1$ is applied to the NS protocol and in the second example it is applied to the KSL protocol.

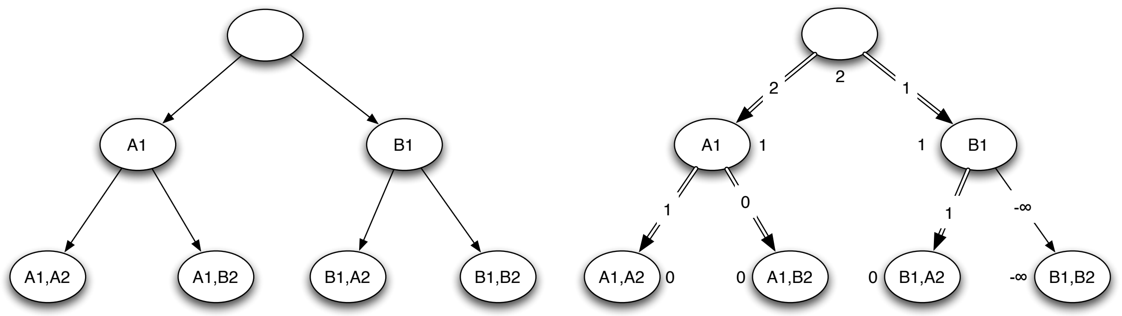
The examples show that $\mathcal{H}_1$ is able to guide the searching algorithm towards promising paths containing attacks. Moreover, a considerable part of the state space is pruned, reducing the number of states to be explored by the searching algorithm.

3.2.2.1 Applying the Heuristic to the Needham-Schroeder Protocol

Let us consider the property ψ_{NS} given in § 2.6.1 as

$$\forall i : A.\ \exists j : B\ (x_j = na_i\ \wedge\ z_i = nb_j) \tag{3.1}$$

Figure 3.1(a) illustrates a portion of the state space of the NS protocol after the first two join transitions when ψ_{NS} is considered. Notice that ψ_{NS} can be falsified in a path where there is a context containing at least one instance of A and no instances of B.



(a) Join transitions of the NS protocol (b) Weighted states in the NS protocol

Figure 3.1: Join transitions and weighted states in the NS protocol

$\mathcal{H}_1$ will assign weights to states and transitions as in Figure 3.1(b). The highlighted paths (those with ’fat’ arrows) are the ones to be explored; the context $\{A_1, A_2\}$ contains the famous Lowe attack [56] reported below:

$$\begin{array}{llll}
1. & A_1 \to I: & \{na_1, A_1\}_{I^+} & \\
2. & A_2 \to I: & \{na_2, A_2\}_{I^+} & \\
3. & I \to A_1: & \{na_1, na_2\}_{A_1{}^+} & \kappa \rhd na_1, na_2 \\
4. & I \to A_2: & \{na_2, na_1\}_{A_2{}^+} & \kappa \rhd na_1, na_2 \\
5. & A_1 \to I: & \{na_2\}_{I^+} & \\
6. & A_2 \to I: & \{na_1\}_{I^+} &
\end{array}$$

The intruder acts as responder for both A_1 and A_2. As a result of step 1 and 2, κ contains na_1 and na_2; enabling the intruder to send messages to A_1 and A_2 at step 3 and 4 respectively. This results into assignments like $z_{A_1} = na_2$ and $z_{A_2} = na_1$; such assignments yield the falsification of (3.1) which requires a nonce generated by B to be assigned to both variables.

The other two highlighted paths contain a similar attack, we report the one with context $\{A_1, B_2\}$.

$$\begin{array}{llll}
1. & A_1 \to I: & \{na_1, A_1\}_{I^+} & \\
2. & I \to A_1: & \{na_1, I\}_{A_1{}^+}, & \kappa \rhd na_1, I \\
3. & A_1 \to I: & \{I\}_{I^+} & \\
4. & I \to B_2: & \{na_1, I\}_{B_2{}^+}, & \kappa \rhd na_1, I \\
5. & B_2 \to I: & \{na_1, nb_2\}_{I^+} & \\
6. & I \to B_2: & \{nb_2\}_{B_2{}^+}, & \kappa \rhd nb_2
\end{array}$$

Again at step 2 and 4, A_1 and B_2 are receiving the identity of intruder instead of nonce by B, resulting into an attack.

It is evident from the Figure 3.1(b) that $\mathcal{H}_1$ assigns appropriate weights to the paths that contain an attack. It is worthy mentioning that the context $\{B_1, B_2\}$ has been labeled $-\infty$, therefore the search will never explore this state that indeed does not lead to attacks.

3.2.2.2 Applying the Heuristic to the KSL protocol

We consider the analysis of (the second phase of) KSL [52], done in [43]. The protocol provides repeated authentication and has two phases; in the first phase (i) a trusted server S generates a session key kab to be shared between A and B, and (ii) B generates the *ticket* $\{Tb, A, kab\}_k$ for A (where Tb is a timestamp and k is known only to B).

In the second phase, A uses the ticket (until it is valid) to repeatedly authenticate herself to B without the help of S. The second phase can be specified as follows:

$$\begin{array}{lll} 1. & A \rightarrow B: & na, \{Tb, A, kab\}_k \\ 2. & B \rightarrow A: & nb, \{na\}_{kab} \\ 3. & A \rightarrow B: & \{nb\}_{kab} \end{array}$$

A sends a fresh nonce na and the ticket to B that accepts the nonce challenge and sends nb together with the cryptogram $\{na\}_{kab}$ to A. In the last message, A confirms to B that she got kab.

In *cIP*, A and B can be represented as follows:

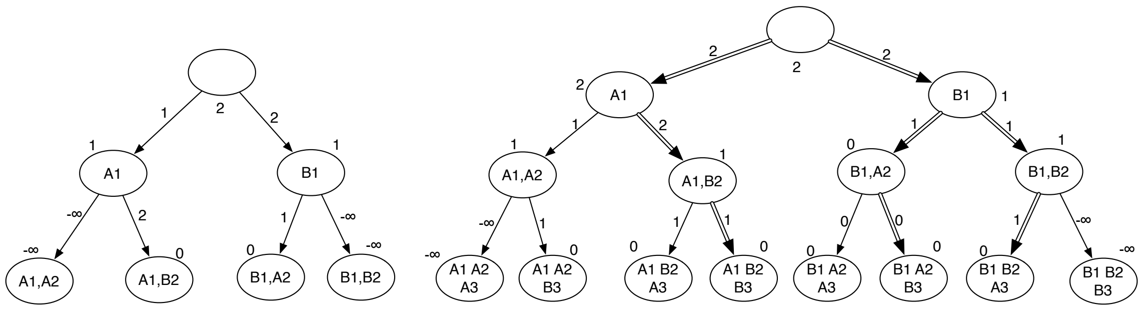
$$\begin{array}{llll} A:(b,sk,tk)[& out(na,\{b,A,sk\}_{tk}). & B:(a,sk,tk)[& in(?x,\{B,a,sk\}_{tk}). \\ & in(?y,\{na\}_{sk}). & & out(nb,\{x\}_{sk}). \\ & out(\{y\}_{sk})] & & in(\{nb\}_{sk})] \end{array}$$

(where for simplicity the timestamp generated by B is substituted by his identity). Authentication is based on the mutually exchanged nonces, and formalised as follows:

$$\psi_{KSL} = \forall l : B.\ \forall j : A.(b_j = B_l \wedge a_l = A_j \rightarrow x_l = na_j \wedge y_j = nb_l)$$

which reads any pair of properly connected "partners" B_l and A_j $(b_j = B_l \wedge a_l = A_j)$

eventually exchange the nonces na_j and nb_l.



(a) 2 principals (b) 3 principals

Figure 3.2: Join transitions of the KSL protocol

Figures 3.2(a) and 3.2(b) depicts the weighted join transitions for 2 and 3 principal instances respectively. The verification with 2 principal instances reports no attack and the conclusion can be derived by just exploring half of the state space (the context $\{A_1,A_2\}$ and $\{B_1,B_2\}$ are labeled $-\infty$; see Figure 3.2(a)). In case of 3 principal instances the attacks are found in highlighted paths (those with 'fat' arrow in Figure 3.2(b)). $\mathcal{H}_1$ assigns appropriate weights to such paths and two states are labeled $-\infty$ referring to the fact that they will never be explored.

3.3 Another Heuristic

We introduce a second heuristic called $\mathcal{H}_2$ that relies on security formulae and open variables. The heuristic $\mathcal{H}_2$ requires two inputs, a state $\langle C, \chi, \kappa \rangle$ and the *disjunctive normal form* (DNF, cf., Definition. 5) of the negation of a formula and returns an integer.

Heuristic $\mathcal{H}_2$ relies on the (quantifier free part of the) security formula. For each state, the heuristic parses the security formula until an atomic formula is found. Specifically, $\mathcal{H}_2$ searches for atomic formulae containing open variables: the state gets the value depending on the way open variables is instantiated in the state. The value of a state is a higher if the assignment of open variables complies with the atomic formula specification, lower otherwise.

Definition 5 (Disjunctive Normal Form)**.** *A formula ϕ, is in* Disjunctive Normal Form *if it is of the form $\phi \equiv \phi_1 \vee \ldots \vee \phi_n$ where each ϕ_i has the form $\psi_{1,i} \wedge \ldots \wedge \psi_{n,i}$ and each $\psi_{j,i}$ is an atomic formula or a negation of an atomic formula.*

Before defining $\mathcal{H}_2$ we introduce some auxiliary functions.

Definition 6. *Given a state $s = \langle C, \chi, \kappa \rangle$, a formula ϕ of the form $\phi_1 \wedge \ldots \wedge \phi_n$ where each ϕ_i is in DNF, the function $\mathcal{H}_\wedge(s,\phi)$ is defined as*

$$\mathcal{H}_\wedge(s,\phi) = \begin{cases} \max\limits_{i=1,\ldots,n} \mathcal{H}_2(s,\phi_i) \ , & \mathcal{H}_2(s,\phi_i) \geq 0 \textit{ for each } \phi_i \\ \min\limits_{i=1,\ldots,n} \mathcal{H}_2(s,\phi_i) \ , & otherwise \end{cases}$$

In order to satisfy ϕ, a state must satisfy each ϕ_i. Therefore, $\mathcal{H}_\wedge(s,\phi)$ evaluates each ϕ_i and returns the maximum value if every invocation results into a positive value otherwise it returns the minimum value (and hence a negative value). Therefore a positive heuristic value for a state indicates that the state satisfies ϕ and a negative value otherwise.

Definition 7. *Given a state $s = \langle C, \chi, \kappa \rangle$ and a formula ϕ of the form $x = m$, the function $\mathcal{H}_=(s,\phi)$ is defined as*

$$\mathcal{H}_=(s,\phi) = \begin{cases} \nu_{s,\phi} \ , & \chi(x) = m \textit{ or } (\chi(x) = x\ [\kappa] \textit{ and } \kappa \rhd m) \\ -1 \ , & \chi(x) \neq m \textit{ or } (\chi(x) = x\ [\kappa] \textit{ and } \kappa \not\rhd m) \\ 0 \ , & \chi(x) = \perp \end{cases}$$

where $\nu_{s,\phi}$ is the number of open variables in s and ϕ.

$\mathcal{H}_=(s,\phi)$ ranks high states satisfying ϕ while states violating ϕ are assigned -1 and must be pruned. The final case in $\mathcal{H}_=(s,\phi)$ deals with a state that does not instantiate

x. Since, no hint is available to evaluate the state therefore a neutral value such is 0 is assigned to it.

Definition 8. *Given a state $s = \langle \mathcal{C}, \chi, \kappa \rangle$ and a formula ϕ of the form $x \neq m$ the function $H_{\neq}(s,\phi)$ is defined as*

$$\mathcal{H}_{\neq}(s,\phi) = \begin{cases} \nu_{s,\phi}\ , & \chi(x) \neq m \textit{ or } (\chi(x) = x\ [\kappa] \textit{ and } \kappa \not\rhd m) \\ -1\ , & \chi(x) = m \\ \nu_{s,\phi} - 1\ , & \chi(x) = x\ [\kappa] \textit{ and } \kappa \rhd m \\ 0\ , & \chi(x) = \perp \end{cases}$$

where $\nu_{s,\phi}$ is the number of open variables in s and ϕ.

Cases 1, 2, and 4 in $\mathcal{H}_{\neq}(s,\phi)$ follow the same intuition as for $\mathcal{H}_{=}(s,\phi)$ that a state satisfying ϕ should be ranked high. Noteworthy, case 3 evaluates a state where x is mapped symbolically. Recall that a symbolic variable is not instantiated immediately (cf.,§ 2.6.1.2). Since such state can neither be pruned not ranked high, $\mathcal{H}_{\neq}$ lowers its heuristic value by decrementing $\nu_{s,\phi}$.

We now define $\mathcal{H}_2$ that depends on functions given in Definition 6, 7, and 8.

Definition 9 (Heuristic $\mathcal{H}_2$)**.** *Given a state $s = \langle \mathcal{C}, \chi, \kappa \rangle$ and a formula ϕ the heuristic function $\mathcal{H}_2$ is defined as*

$$\mathcal{H}_2(s,\phi) = \begin{cases} \max\limits_{i=1,\ldots,n} \mathcal{H}_2(s,\phi_i), & \phi \equiv \phi_1 \vee \ldots \vee \phi_n \\ \mathcal{H}_\wedge(s,\phi)\ , & \phi \equiv \phi_1 \wedge \ldots \wedge \phi_n \\ \mathcal{H}_=(s,\phi)\ , & \phi \equiv\ x = m \\ \mathcal{H}_\neq(s,\phi)\ , & \phi \equiv\ x \neq m \\ 1\ , & \phi \equiv true \\ -1\ , & \phi \equiv false \\ 0\ , & otherwise \end{cases}$$

The recursive application of $\mathcal{H}_2$ over the formula ϕ along with s determines the heuristic value to be assigned to the state. We discuss each case and describe why a particular value has been chosen.

Case 1 is applied when ϕ is a disjunction, i.e., $\phi \equiv \phi_1 \vee \ldots \vee \phi_n$. Each disjunct ϕ_i is evaluated individually and the maximum value is returned. Intuitively, a state satisfies ϕ when it satisfies at least one ϕ_i. Thus $\mathcal{H}_2(s,\phi)$ returns a positive value if, and only if, ϕ is satisfied. Hence, if $\mathcal{H}_2(s,\phi) < 0$ then the state s can be pruned.

Case 2 is used when ϕ is a conjunction, i.e., $\phi = \phi_1 \wedge \ldots \wedge \phi_n$. The state must be model for all the conjuncts and is evaluated by $\mathcal{H}_\wedge$ given in Definition 6.

Cases 3-7 are used when ϕ is an atomic formula. Cases 3 and 4 respectively deals with the atomic formula of the form $x = m$ and $x \neq m$ and are evaluated by the functions $\mathcal{H}_=$ and $\mathcal{H}_\neq$ respectively. The other cases are trivial.

3.3.1 Improving $\mathcal{H}_2$

$\mathcal{H}_2$ can be improved by slightly modifying $\mathcal{H}_\wedge$ so that the values returned on sub-formulas are added up in order to guarantee that a state satisfying more conjuncts is preferred. The drawback of this approach is that the new heuristic may produce two different values for a state against logically equivalent formulae. For instance, *true* and *true* $\wedge$ *true* would be evaluated to 1 and 2 respectively. However, we conjecture that the new heuristic is correct (i.e., no attacks are eliminated when pruning the state space). In fact, when a state is pruned the minimal value is returned as per case 2 of Definition. 6.

3.3.2 Evaluation of $\mathcal{H}_2$

It is important to mention that $\mathcal{H}_1$ is generally more efficient than $\mathcal{H}_2$ as $\mathcal{H}_1$ inspects the quantification (over principal instances) in a security formula and for each state tries to establish if such principal (instances) have joined the context. On the other hand, $\mathcal{H}_2$ exploits open variables (i.e., symmetric keys and identities) and assigns a weight to a state depending on how such variables have been instantiated in the state. We conjecture that $\mathcal{H}_2$ can perform better than $\mathcal{H}_1$ when the security formula specifies symmetric keys in addition to principal identities.

We illustrate this with the help of the following example. We have designed two security formulae for the ISO protocol. Both formulae specify principal identities but only one specifies symmetric keys.

The ISO protocol belongs to a family of authentication mechanisms presented in [50]. It performs unilateral authentication of a principal A to a principal B. The informal specification for ISO is:

$$\begin{aligned} 1.\, B \rightarrow A:&\quad nb,\ \text{Text1}\\ 2.\, A \rightarrow B:&\quad \text{Text3}, \{nb, B, \text{Text2}\}_{kab} \end{aligned}$$

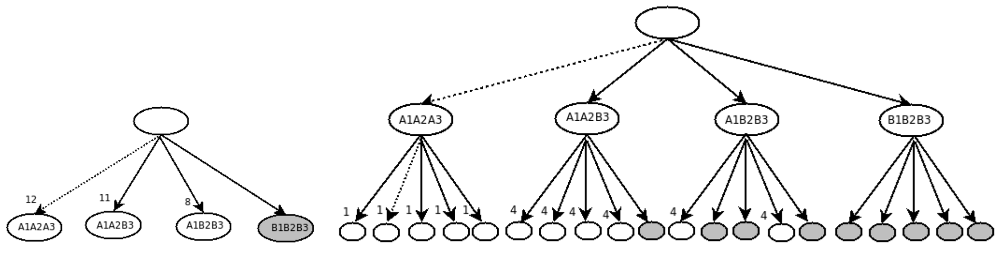
A and B share a symmetric key, called *kab*. *B* sends to *A* a random number (nb) along with a text field. For verification purposes, the text field is represented by a nonce. *A* sends back a cryptogram containing *nb* and $B's$ identity, thus *B* believes that he is talking to A.

$$A : (b, sk)[\quad in((?r, ?td)). \quad out(na, \{((r,b), nc)\}_{sk})] \qquad B : (sk)[\quad out(nb, nd). \quad in(?ta, \{(nb, B), ?tc)\}_{sk})]$$

The protocol is verified against a security formula that specifies symmetric keys and is given as

$$\psi_{ISO_1} = \forall l : A.\ \exists o : B.\ b_l = B_o \implies sk_l = sk_o$$

which states that if A is connected to B then they share the same symmetric key.



(a) $\mathcal{H}_1$ (b) $\mathcal{H}_2$

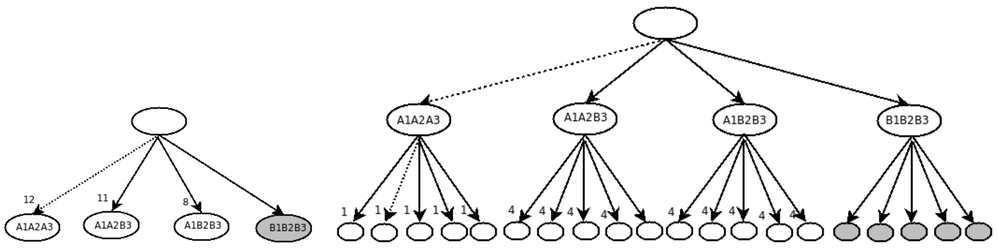
Figure 3.3: Partial state space (weighted) for the ISO with ψ_{ISO_1}

Figure 3.3(a) and Figure 3.3(b) respectively show the weights assigned to states by $\mathcal{H}_1$ and $\mathcal{H}_2$ for ψ_{ISO_1}. The path leading to an attack is depicted as a dotted line and shaded states represent the pruned states. Note that in Figure 3.3(a), the context $\{B_1, B_2, B_3\}$ is shaded representing the fact that this state will not be explored further for analysis. However, $\mathcal{H}_2$ as shown in Figure 3.3(b) can prune all the traces emerging from $\{B_1, B_2, B_3\}$ and additionally prunes one trace from the context $\{A_1, A_2, B_3\}$ and three traces from the context $\{A_1, B_2, B_3\}$. Thus $\mathcal{H}_1$ reduces the state space by 6% while $\mathcal{H}_2$ reduces the state space by 16% (see § 5.1.4 for actual numbers).

We now consider a formula which specifies identities and nonces instead of symmetric keys.

$$\psi_{ISO_2} = \forall l : A.\ \exists o : B.\ b_l = B_o \implies r_l = nb_j$$

which states that if A is connected to B then the nonce received by A, r_l is the one sent by B, nb_j.



(a) $\mathcal{H}_1$ (b) $\mathcal{H}_2$

Figure 3.4: Partial state space (weighted) for the ISO with ψ_{ISO_2}

Figure 3.4 shows the state space as evaluated by $\mathcal{H}_1$ and $\mathcal{H}_2$ for ψ_{ISO_2}. Note that the efficiency of $\mathcal{H}_2$ and $\mathcal{H}_1$ will be approximately the same as $\mathcal{H}_2$ does not prune any additional traces from the contexts $\{A_1, A_2, B_3\}$ and $\{A_1, B_2, B_3\}$. The additional pruning in the case of ψ_{ISO_1} is due to the specification of symmetric keys and thus in the case of ψ_{ISO_2} both heuristics give the same efficiency.

3.4 Combining $\mathcal{H}_1$ and $\mathcal{H}_2$

We have combined the heuristic functions $\mathcal{H}_1$ and $\mathcal{H}_2$ to obtain a *composite heuristic* [66] called $\mathcal{H}_3$. A composite heuristic has more than one heuristic functions at its disposal permitting it to choose the most appropriate function for evaluating each state. The heuristic $\mathcal{H}_3$ applies $\mathcal{H}_1$ for states that are generated due to join transitions and $\mathcal{H}_2$ to the states that are generated due to instantiation of open variables.

Definition 10. *Given a state s, a transition t from s to* $s' = \langle \mathcal{C}', \chi', \kappa' \rangle$ *in* $s\Upsilon$, *and a formula* ϕ, *the heuristic function* $\mathcal{H}_3$ *is given by*

$$\mathcal{H}_3(s', \phi) = \begin{cases} \mathcal{H}_s(s', \phi)\ ,\ \forall\, A \in \mathcal{C}',\ \forall\, x \in X_A :\ \ \chi(x) = \perp \\ \mathcal{H}_2(s', \phi)\ ,\ \exists\, A \in \mathcal{C}', \exists\, x \in X_A :\ \ \chi(x) \neq \perp \end{cases}$$

where, given a principal A, we let X_A denote its set of open variables.

Thus heuristic function $\mathcal{H}_3$ is invoked if the state s' is the outcome of a *join* transition. The function $\mathcal{H}_3$ takes the state s' and ϕ as input and checks s' to decide the appropriate heuristic function for evaluating the state s'. If the open variables associated with the principal instances particiapting in the context are not yet initialized, it invokes the heuristic $\mathcal{H}_s$. Otherwise, $\mathcal{H}_3$ invokes the heuristic function $\mathcal{H}_2$.

3.5 Properties of the Heuristics

In this section we present some comments on admissibility and optimality of our heuristics.

3.5.1 Admissibility

Admissibility ensures that heuristic estimates for a state are always lower than the actual cost to reach the goal state [54]. Admissibility is therefore desirable for *estimation-based* heuristic functions to ascertain the optimality of the obtained solution. Our heuristic functions ($\mathcal{H}_1$, $\mathcal{H}_2$, $\mathcal{H}_3$) are defined as *evaluation based* functions that evaluate the *likeliness* for the state (and transitions) to lead to an attack. The weights assigned by the heuristics to states do not correspond to the proximity to a target state. Since our heuristics do not represent the *cost* to reach the goal node therefore admissibility in this case cannot be applied.

3.5.2 Optimality

We discuss two different notions of optimality and comment about optimality of our heuristics in both cases.

An optimal solution is said to be obtained when the search finds the *best* solution from an available set of feasible solutions. In terms of directed model checking this can be put as having *different possible* goal states and heuristics guiding the search towards the best goal state thus yielding the optimal solution. In the case of security protocols the goal is an "attack", namely a state that violates the security property. Typically, it is very hard to compare the importance of different attacks as the violation of a property may be due to many causes (as for the NS Example in § 3.2.2.1). Therefore, isolating any solution (i.e., an attack) as an optimal solution is hard. Thus finding an attack *quickly* is what is desired and that can be achieved by using evaluation based heuristics such as $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$.

Another criteria for optimality is the length of the path leading to an attack state. This has an additional advantage of producing shorter counter examples in addition to finding goals quickly. However, this notion of optimality is only possible with *admissible* heuristics. As said earlier, admissibility applies only to estimation-based heuristics thus our heuristics do not achieve this notion of optimality. Nevertheless, we envisage the importance of estimation-based heuristics for security protocol verification as they generate shorter counter examples. However, this thesis does not deal with estimation based heuristics for security protocol verification.

3.6 Incorporating Heuristics into a Searching Algorithm

The heuristics defined in § 3.2, § 3.3, and § 3.4 are incorporated into a searching algorithm which is a hybrid algorithm as the initial phase of verification is informed while the second phase is done with a conventional DFS. An intuition of how the algorithm works is given in Figure 3.5 where join transitions (the state space consisting of states ABCDEF) are explored according to their heuristic values while the search

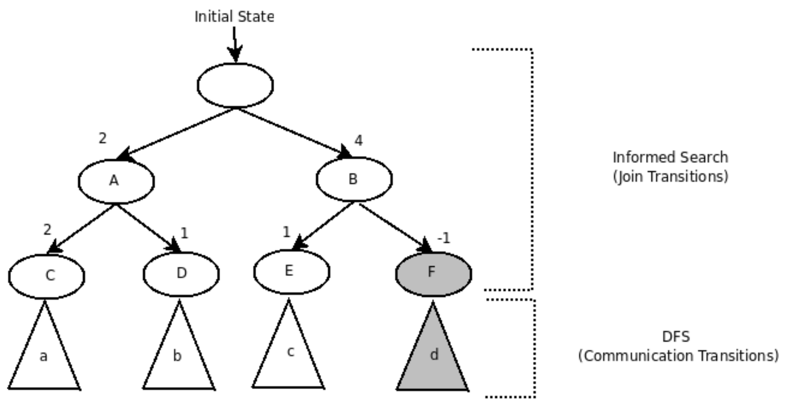


Figure 3.5: Combining Heuristic Search and Conventional DFS

in the remaining state space (abcd) remains uninformed.

We now justify the need for a hybrid algorithm and discuss if heuristics can be applied in the remaining state space.

Heuristic $\mathcal{H}_1$ is only applied to join transitions. Technically, it cannot be extended to communication transitions as it exploits the information available only in join transitions.

Heuristic $\mathcal{H}_2$ can be adapted for communication transitions. However, one of the major issues in heuristic searches is the cost of computing the heuristic function. Thus, heuristics should only be applied if their cost is less than the efficiency obtained. We conjecture that more efficiency can be achieved by applying heuristics in the join transitions. More precisely, maximum branching occurs due to join transitions because all possible combinations of protocol participants and instantiations of open variables must be tried. Thus more efficiency is obtained if heuristic search efficiently explores join transitions. Nevertheless, in future, we intend to extend our hybrid algorithm to a complete DMC algorithm by adapting $\mathcal{H}_2$ for communication transitions.

Chapter 4

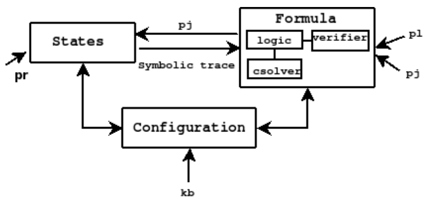
Architecture and Implementation of $\mathcal{H}$-ASPASyA

The extension of $\mathcal{A}$SPASyA with our heuristics will hereafter be referred to as $\mathcal{H}$-ASPASyA. Actually, $\mathcal{H}$-ASPASyA can explore states according to the weights assigned to them by either $\mathcal{H}_1$, $\mathcal{H}_2$, or $\mathcal{H}_3$, respectively referred to as $\mathcal{H}_1$-ASPASyA, $\mathcal{H}_2$-ASPASyA, and $\mathcal{H}_3$-ASPASyA.

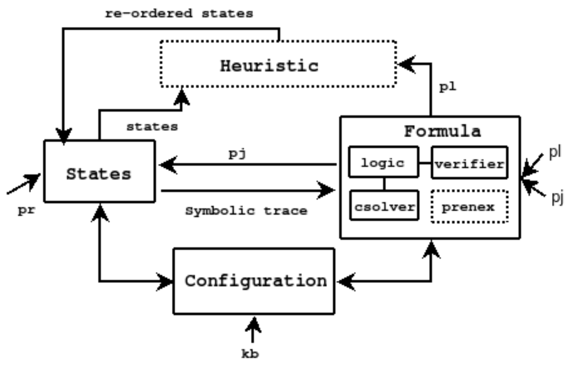
The architecture for $\mathcal{A}$SPASyA and $\mathcal{H}$-ASPASyA is respectively shown in Figure 4.1(a) and 4.1(b). Note that in Figure 4.1(b), `Heuristic` is the abstract representation for the module responsible for assigning weights to states. The actual structure of the module `Heuristic` varies with heuristic $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ and will be detailed in appropriate sections. In § 4.1, we describe the architecture of $\mathcal{H}$-ASPASyA and in § 4.2 we illustrate the details of its implementation.

Before describing the architecture of $\mathcal{H}$-ASPASyA it is worth to briefly comment on the architecture of $\mathcal{A}$SPASyA (see [9] for a detailed description).

The main modules of $\mathcal{A}$SPASyA are in Figure 4.1(a) and are called `States`, `Configuration`, and `Formula`. They have been extended as shown in Figure 4.1(b)



(a) $\mathcal{A}$SPASyA



(b) $\mathcal{H}$-ASPASyA

Figure 4.1: Architecture of $\mathcal{A}$SPASyA and $\mathcal{H}$-ASPASyA

to accommodate a new module, called `Heuristic`, and a new component added to the module `Formula` called `prenex`.

The module `Configuration` serves as the basic building block for computing states and has three components namely `context`, `assignment`, and `knowledge` to represent a state $\langle \mathcal{C}, \chi, \kappa \rangle$. The module `States` invokes `Configuration` to perform a transition so to update `context`, `assignment`, and `knowledge` accordingly and return the next configuration to `States`. The module `States` has two components namely `join` and `step`. The former implements join transitions and the latter deals with communication transitions. The module `Formula` consists of the components `logic`, `verifier`, and `csolver`. The first component converts a $\mathcal{PL}$ formula into DNF, the second checks whether a given configuration is a model for the formula, and the latter deals with constraints on symbolic variables.

The input expected by $\mathcal{A}$SPASyA consists of (i) the protocol formalised in *cIP* (say `pr`), (ii) the security property (say `pl`), (iii) the join formula (say `pj`) in $\mathcal{PL}$, (iv) a finite set of messages representing the initial knowledge of the intruder (say `kb`), and (v) the maximum number of instances (say `m`) that can join a session. From the initial input, $\mathcal{A}$SPASyA first generates the join transitions by invoking the module `States`; which in turn invokes `configuration` to update `context`, `assignment`, and

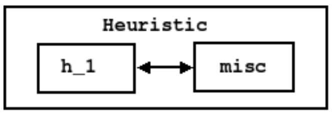


Figure 4.2: Module `Heuristic` for $\mathcal{H}_1$-ASPASyA

`knowledge`. The module `States` now interacts with the module `Formula` so to filter out the states that do not satisfy `pj` and expand those satisfying it with communication transitions. The module `States` invokes `configuration` on each context to fire communication transitions according to the semantics of *cIP*. Configurations are then forwarded to `Formula` to check the presence of possible attacks.

The modular structure of $\mathcal{A}$SPASyA has been instrumental to straightforwardly apply our modifications to accommodate heuristics; the implementation remains mostly unaffected and only requires the introduction of new modules and minor changes to the existing ones.

4.1 Architecture of $\mathcal{H}$-ASPASyA

4.1.1 $\mathcal{H}_1$-ASPASyA

The module `Heuristic` assigns weights to states and transitions depending on the $\mathcal{PL}$ security formula. As shown in Figure 4.2, `Heuristic` contains two components namely `h_1` and `misc`. The component `h_1` assigns weights and provides the core implementation for the heuristic functions $\mathcal{H}_s$ and $\mathcal{H}_t$ defined in § 3.2, while `misc` contains miscellaneous functions that are used by the component `h_1`. Further, the module `Formula` has been modified so as to have an additional component called `prenex`, that converts a $\mathcal{PL}$ formula into an equivalent PNF. The other components of the module `Formula` namely `Logic`, `verifier`, and `csolver` remain unaffected.

The Architecture of $\mathcal{H}_1$-ASPASyA has been designed so to pre-process join transitions and weight them. Weighted join transitions are generated by the component h_1 which relies on prenex and States to respectively transform pl in PNF and to generate join transitions as in $\mathcal{A}$SPASyA.

However, the data structure used by $\mathcal{A}$SPASyA for representing states is rather more complex than what the heuristic needs; for the sake of simplicity and to limit the changes to $\mathcal{A}$SPASyA to a minimum, we decided to use a simpler data structure in the module Heuristic. The states in Heuristic are represented with contexts only instead of $\langle C, \chi, \kappa \rangle$. This requires to map back the states in Heuristic to those in States so that the latter generates the communication transitions after acquiring weights. Therefore, the component misc yields a functionality to *match* the states of Heuristic with those of States. Intuitively, there is a match between two such states when their corresponding contexts have same principal instances. The weighted states are then re-ordered and given back to States to explore them further.

4.1.2 $\mathcal{H}_2$-ASPASyA

$\mathcal{H}_2$-ASPASyA refers to extension of $\mathcal{A}$SPASyA with the heuristic $\mathcal{H}_2$. The module Heuristic for $\mathcal{H}_2$-ASPASyA contains two components i.e., h_2 and h_var, as shown in Figure 4.3. The former contains the implementation for the heuristic function $\mathcal{H}_2$

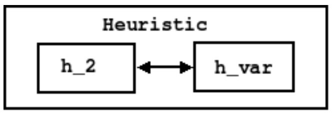


Figure 4.3: Module Heuristic for $\mathcal{H}_2$-ASPASyA

presented in § 3.3 and the latter contains some auxiliary functions used by $\mathcal{H}_2$ and described below.

As in $\mathcal{A}$SPASyA the module States generates join transitions; followed by generation of new states by instantiating open variables with possible values. Afterwards, States invokes the function H_2 in the component h_2 to weight the new states. The component h_2 interacts with Formula to obtain DNF of $\mathcal{PL}$ formulae. It then weights the states accordingly and passes them to h_var for further processing. More precisely, some states are pruned and remaining ones are ordered according to the weights assigned to them by h_2. Likewise in h_1, the ordered set of states is then returned to the module States to be model checked.

4.1.3 $\mathcal{H}_3$-ASPASyA

$\mathcal{H}_3$-ASPASyA is the name given to the extension of $\mathcal{A}$SPASyA with $\mathcal{H}_3$ that combines $\mathcal{H}_1$ and $\mathcal{H}_2$. The module Heuristic as shown in Figure 4.4, therefore contains components h_1 and misc for implementing $\mathcal{H}_1$, h_2 and h_var for implementing $\mathcal{H}_2$, and a new component h_3.

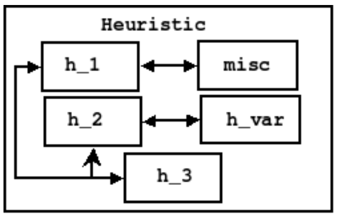


Figure 4.4: Module Heuristic for $\mathcal{H}_3$-ASPASyA

The module States first generates states where required number of instances have joined the context and passes them to h_3 which invokes h_1 to weight them. After weighting, pruning and re-ordering, the states are returned to States which expands them by instantiating open variables. The resulting set of states is handed over to h_3 that invokes h_2 to evaluate them. The re-ordered set of states is finally returned to the module States.

4.2 Algorithms and Implementation for $\mathcal{H}$-ASPASyA

$\mathcal{A}$SPASyA is implemented in Ocaml (version 3.06). Ocaml is a strongly typed functional language that supports recursive data structures and pattern matching. Exploiting such features, the state space of $\mathcal{H}$-ASPASyA is implemented as a tree-like structure that can be traversed using pattern matching. We report the algorithm for $\mathcal{H}_1$-ASPASyA in § 4.2.1, followed by algorithm for $\mathcal{H}_2$-ASPASyA in § 4.2.2, and finally the algorithm for $\mathcal{H}_3$-ASPASyA is in § 4.2.3.

We first detail the data structures used and then report the algorithm for $\mathcal{H}_1$-ASPASyA.

The generation of join transitions in $\mathcal{H}_1$-ASPASyA is materialised using a tree data structure defined by using sum types (i.e., Ocaml union types) and tuples. Ocaml pattern matching mechanisms are used for traversal of the following data types

$$\begin{aligned} \texttt{type weight} &= \texttt{Int of int} \mid -\texttt{Infty} \\ \texttt{type princlist} &= (\texttt{string} * \texttt{int})\ \texttt{list} \\ \texttt{type node} &= \texttt{Node of}\ (\texttt{weight} * \texttt{princlist}) \end{aligned} \tag{4.1}$$

where a `weight` can be an integer or a special value $-$`Infty` to represent $-\infty$, `princlist` is a list of principal instances representing contexts (an instance is modelled as a pair whose first component is the identity of principal and the second component is its index), and a `node` is a pair of `weight` and `princlist`.

The type `wt` of a weighted tree is defined as

$$\texttt{type wt} = \texttt{WT of node} * (\texttt{weight} * \texttt{wt})\ \texttt{list}$$

A weighted tree (`wt`) is a pair of a `node` and a list of tuples representing transitions associated to that node. Each tuple contains the weight of the transition and a weighted tree `wt` for representing subtree attached to that transition.

4.2.1 Algorithm for $\mathcal{H}_1$-ASPASyA

The computational steps for $\mathcal{H}_1$-ASPASyA are sketched in Algorithms 2 and 3. Algorithm 2 is basically a slight modification of $\mathcal{A}$SPASyA and shows that at step 3 we invoke `compute_H`$_1$ function reported in Algorithm 3.

Hereafter, we denote states of $\mathcal{A}$SPASyA as $\langle \mathcal{C}, \chi, \kappa \rangle$ and use the data types in (4.1) to indicate the structure of $\mathcal{H}_1$-ASPASyA. We demonstrate the algorithms with the help of our running example of the NS.

Algorithm 2 $\mathcal{H}$-ASPASyA – **Input**: `pr`, `pl`, `pj`, `kb`, `m`

```
1: n ← $\langle \emptyset, \emptyset, \mathtt{kb} \rangle$
2: l ← a list of states $\langle \mathcal{C}, \chi, \kappa \rangle$
3: ol ← Compute_H$_1$(l)
4: ll ← for each state in ol generate a list of states with open variables
        instantiated to possible values.
5: For each context start communication between participants.
6: Verify each configuration according to pl
```

The first step of Algorithm 2 creates the root node with the empty context $\emptyset$, the empty substitution $\emptyset$, and initial knowledge `kb`. The next step generates a list `l` of states each of which can have at most `m` principal instances in the context. We invoke the function `Compute_H`$_1$ at step 3 to compute the weights of the states generated at step 2. Step 4 of Algorithm 2 computes the mappings of open variables and generates new configurations. The final steps generate communication transitions and verify them according to $\mathcal{PL}$.

We continue commenting on Algorithm 3 described by the following ocaml pseudocode.

Algorithm 3 Compute_H_1 – **Input**: l, pr, pl, m,

```
1: let t ← gen_tree (m, pr) in
2:    let ht ← $H_1$ (t, pnf (pl)) in
         let wl ← ext_context (ht) in
3: let wc ← find_w (l wl) in
      let swc ← sort_w (wc) in
         sc ← re_order (swc)
```

In step 1 of Algorithm 3, gen_tree (in component h_1) is invoked with the formalised protocol pr to return the contexts (with weights initialised to zero) obtained by joining m principals (join transitions are generated using the data structures given in (4.1)).

For the NS, the tree generated with m = 2 is given in Figure 4.5 where (on the left-

```
WT (State (0, [ ])
  [(0,
    WT (State (0, [ A1 ]),
    [(0,WT (State (0, [A1;A2]),[ ]));
      (0,WT (State (0, [A1;B2]),[ ]))]));
  (0,
    WT (State (0,[ B1 ]),
    [(0,WT (State (0,[B1;A2]),[ ]));
      (0,WT (State (0, [B1;B2]),[ ]))]))])
```

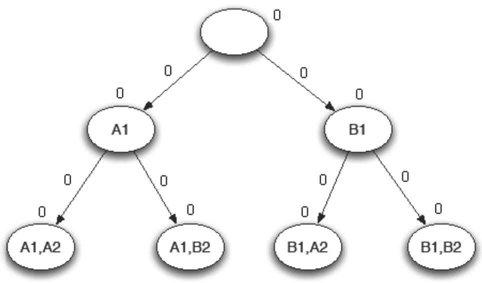


Figure 4.5: Join transitions for NS with weights initialised to zero

hand-side) the root node has empty list of principals and two associated transitions; each transition has a weight initialised to zero and an associated subtree. The left subtree adds a principal instance A_1 to the context inherited from parent node and the right subtree adds B_1 to its parent context. The process is repeated until the required number of instances have joined the context.

At step 2, the tree t is weighted using H_1 defined in the component h_1 of module Heuristic along with prenex normal form of the security property pnf(pl) (function pnf is in prenex). For our example, H_1 will return the tree given in Figure 4.6 for

```
WT (State (2, [ ])
  [(2,
    WT (State (1, [ A1 ]),
    [(1,WT (State (0, [A1;A2]),[ ]));
      (0,WT (State (0, [A1;B2]),[ ]))]));
  (1,
    WT (State (1,[ B1 ]),
    [(1,WT (State (0,[B1;A2]),[ ]));
      (−∞,WT (State (−∞, [B1;B2]),[ ]))]))])
```

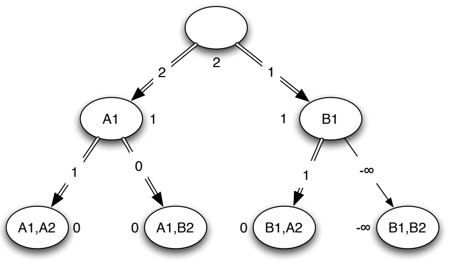


Figure 4.6: Weighted join transitions for NS

NS with two instances. Function $\texttt{H}_1$ inspects the formula and the context as described before. The function ext_context extracts the weighted contexts (those with m instances) from the weighted tree, namely it returns the list of weighted leaves of the tree ht (the weights are obtained by adding all the weights on the path starting from the root node and terminating at leaf). Applying function ext_context to the tree given in Figure 4.6 yields

$$[(6,\ [\texttt{A}_1;\texttt{A}_2]);(5,\ [\texttt{A}_1;\texttt{B}_2]);(5,\ [\texttt{B}_1;\texttt{A}_2]);(-\infty,\ [\texttt{B}_1;\texttt{B}_2])]$$

The function find_w (in the component misc of Heuristic) takes l and wl as inputs and finds the match for each state in l by comparing its context with the contexts in wl. When a match is found, the weight associated to the context in wl is attached to its corresponding context in l. This modifies the list l so that it now has the tuples of weight and state $\langle \mathcal{C}, \chi, \kappa \rangle$.

$$[(6,(\ [\texttt{A}_1;\texttt{A}_2],\ [\texttt{I},\texttt{I}^+,\texttt{I}^-,\texttt{A}_1,\texttt{A}_1^+,\texttt{A}_2,\texttt{A}_2^+],\ [\,]));$$
$$(5,(\ [\texttt{A}_1;\texttt{B}_2],\ [\texttt{I},\texttt{I}^+,\texttt{I}^-,\texttt{A}_1,\texttt{A}_1^+,\texttt{B}_2,\texttt{B}_2^+],\ [\,]));$$
$$(\ -\infty,(\ [\texttt{B}_1;\texttt{B}_2],\ [\texttt{I},\texttt{I}^+,\texttt{I}^-,\texttt{B}_1,\texttt{B}_1^+,\texttt{B}_2,\texttt{B}_2^+],\ [\,]))]$$

It is worth noting that $\mathcal{A}$SPASyA has an optimization where contexts that are permutations of each other are represented just once. For example in the NS protocol the contexts $\{\mathtt{A}_1,\mathtt{B}_2\}$ and $\{\mathtt{B}_1,\mathtt{A}_2\}$ are permutation of each other and it is sufficient to explore any one of them. Thus before matching `l` with `wl`, the states that are permutations of other states are discarded so that the numbers of states in `l` and `wl` remain the same.

The function `sort_w` sorts the list `wc` on weights in descending order yielding

$$\begin{array}{ll} [& (6,(\,[\mathtt{A}_1;\mathtt{A}_2],\ [\mathtt{I},\mathtt{I}^+,\mathtt{I}^-,\mathtt{A}_1,\mathtt{A}_1^+,\mathtt{A}_2,\mathtt{A}_2^+],\ [\,]));\\ & (5,(\,[\mathtt{A}_1;\mathtt{B}_2],\ [\mathtt{I},\mathtt{I}^+,\mathtt{I}^-,\mathtt{A}_1,\mathtt{A}_1^+,\mathtt{B}_2,\mathtt{B}_2^+],\ [\,]));\\ & (-\infty,(\,[\mathtt{B}_1;\mathtt{B}_2],\ [\mathtt{I},\mathtt{I}^+,\mathtt{I}^-,\mathtt{B}_1,\mathtt{B}_1^+,\mathtt{B}_2,\mathtt{B}_2^+],\ [\,]))] \end{array}$$

Finally the function `re_order` removes the first element of each tuple i.e., the weight. It also removes the context(s) that have received the weight of $-\infty$ giving

$$\begin{array}{ll} [& (\,[\mathtt{A}_1;\mathtt{A}_2],\ [\mathtt{I},\mathtt{I}^+,\mathtt{I}^-,\mathtt{A}_1,\mathtt{A}_1^+,\mathtt{A}_2,\mathtt{A}_2^+],\ [\,]);\\ & (\,[\mathtt{A}_1;\mathtt{B}_2],\ [\mathtt{I},\mathtt{I}^+,\mathtt{I}^-,\mathtt{A}_1,\mathtt{A}_1^+,\mathtt{B}_2,\mathtt{B}_2^+],\ [\,])\,] \end{array}$$

4.2.2 Algorithm for $\mathcal{H}_2$-ASPASyA

Algorithm 4 and 5 specify computation steps required for implementing heuristic $\mathcal{H}_2$. Algorithm 4 is an adaptation of $\mathcal{A}$SPASyA where at step 4, the heuristic function $\mathtt{Compute\_H_2}$ is invoked to weight the states generated at step 3 of Algorithm 4.

Algorithm 4 $\mathcal{H}_2$-ASPASyA – **Input**: `pr`, `pl`, `pj`, `kb`, `m`

```
1: n ← ⟨∅, ∅, kb⟩
2: l ← a list of states ⟨$\mathcal{C}$, $\chi$, $\kappa$⟩
3: ll ← for each state in l generate a list of states with open variables
        instantiated to possible values.
4: Compute_H2(ll, pl)
5: For each context start communication between participants.
6: Verify each configuration according to pl
```

Algorithm 2 and Algorithm 4 are similar and differ at step 3 and 4. In $\mathcal{H}_1$-

ASPASyA, the heuristic function $\texttt{Compute\_H}_1$ is invoked at step 3 to assign heuristic estimates to the states generated at step 2. Whereas in $\mathcal{H}_2$-ASPASyA, step 3 generates states with open variables instantiated to possible values and are handed over to the heuristic function $\texttt{Compute\_H}_2$ at step 4 to acquire weights. The function $\texttt{Compute\_H}_2$ is given in Algorithm 5; we use our running example of NS protocol to explain each step of the Algorithm 5.

Algorithm 5 $\texttt{Compute\_H}_2$ – **Input**: `l`, `pl`

```
1:  let wl ← $H_2$ (l, pl) in
2:  let pr_l ← remove_w (wl) in
      ol ← re_order (pr_l)
```

Algorithm 5 takes in input a list l generated at step 3 of Algorithm 4 and the security formula pl in DNF. In step 1 of Algorithm 5, `H_2` (in the component `h_2`) returns a list `wl`; the elements of `wl` are list of pairs value-state $(\texttt{v}, \texttt{s})$ representing the value `v` of a state `s` according to $\mathcal{H}_2$ (as specified in Definition 9). For instance, the invocation of $\texttt{H}_2$ for the NS protocol yields

$$[\,[(1,\ ([A_1;A_2],\ \ldots))];\ [(0,\ ([A_1;B_2],\ \ldots))];\ [(-1,\ ([B_1;B_2],\ \ldots))]\,]$$

where the ellipses stand for the mapping of variables and knowledge of the intruder (they are not given because immaterial). At step 2 the states with weight -1 are discarded and each sub-list is re-ordered to give

$$[\,[\,(1,\ ([A_1;A_2],\ \ldots))];\ [(0,\ ([A_1;B_2]\ \ldots))]\,]$$

Note that in the case of NS protocol each sub-list comprises of only one state and therefore re-ordering is not possible. The resultant list is returned to the Algorithm 4, which proceeds by generating new configurations which emerge as a result of communication actions executed by the participants.

4.2.3 Algorithm for $\mathcal{H}_3$-ASPASyA

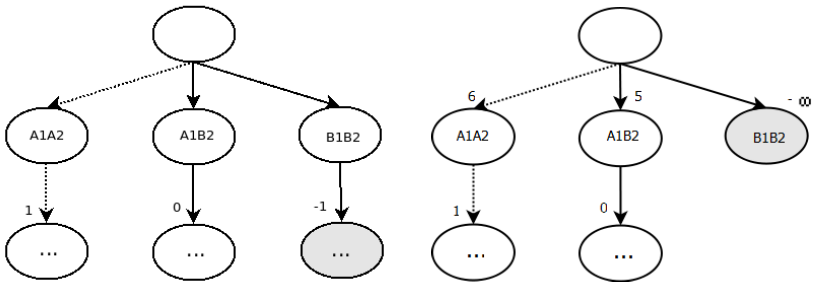
The computational steps for $\mathcal{H}_3$-ASPASyA are listed in Algorithm 6, where both heuristics ($\mathcal{H}_1$ and $\mathcal{H}_2$) have been incorporated in $\mathcal{A}$SPASyA.

Algorithm 6 $\mathcal{H}_3$-ASPASyA – **Input**: pr, pl, pj, kb, m

```
1: n ← ⟨$\emptyset$,$\emptyset$,kb⟩
2: l ← a list of states ⟨$\mathcal{C},\chi,\kappa$⟩
3: Compute_H$_1$(l)
4: ll ← for each state in ol generate a list of states with open variables
       instantiated to possible values.
5: Compute_H$_2$(ll,pl)
6: For each context start communication between participants.
7: Verify each configuration according to pl
```

At step 3 of Algorithm 6, the function Compute_H$_1$ has been invoked. The pseudocode for the function Compute_H$_1$ is reported in Algorithm 3 which weights the list of states l generated at step 2 of Algorithm 6. Each state in the list (of states) returned by compute_H$_1$ is expanded by instantiating open variables. The new list generated is weighted by compute_H$_2$ function reported in Algorithm 5. Afterwards, the states are expanded with communication transitions and verified according to pl.

Figures 4.7(a) and 4.7(b) illustrates the weights assigned to the NS protocol by $\mathcal{H}_2$ and $\mathcal{H}_3$ respectively. Note that Figure 4.7 extends Figure 4.6, where each context has been expanded with possible instantiations of open variables. For the NS protocol, there is only one such possibility therefore each context has only one child node. Note that in Figure 4.7(a), the shaded state is assigned the value -1 and will not be explored further. However, since $\mathcal{H}_1$ prunes the context $\{B_1,B_2\}$ (see Figure 4.7(b)) therefore the combined efficiency of $\mathcal{H}_3$ in the case of NS protocol will not be better than that of $\mathcal{H}_1$.



(a) $\mathcal{H}_2$ (b) $\mathcal{H}_3$

Figure 4.7: Weighted (partial) state space for NS protocol with $\mathcal{H}_2$ and $\mathcal{H}_3$

Chapter 5

Experimental Results and Analysis

The evaluation of the effectiveness of our heuristics has been measured in a number of experiments where a valuable set of protocols have been model checked against a few security properties under different conditions (the complete references for protocols are in Appendix A)[1]. More precisely, the protocols are verified using different join formulae, different security properties, and varying the maximum number of principal instances.

5.1 Results on the State Space

In this section, we present our results for the efficiency obtained in terms of the size of state space. In § 5.1.1 and § 5.1.2, we give the results for the protocols verified with an *arbitrary join* and *constrained join* respectively. In § 5.1.3, we demonstrate the results for a few protocols that have been tested with four instances of the principals. Finally, in § 5.1.4 we give the results for experiments designed specifically for $\mathcal{H}_2$.

The results reported in this section correspond to two capabilities of the heuristics mentioned in § 1.2, i.e., re-ordering and pruning. Re-ordering refers to the directed

[1] $\mathcal{H}$-ASPASyA is available at http://www.cs.le.ac.uk/people/et52/aspasya/h-aspasya.html.

search; more precisely, $\mathcal{H}$-ASPASyA explores the states according to the heuristic values assigned to them and stops as soon as the first attack is found. The number of states explored helps in assessing whether the order in which heuristics put the states is reasonable.

Pruning on the other hand is also helpful when one is interested in discovering all attacks possible on a protocol for the stated security property. Such situation requires exhaustive search of the state space and re-ordering does not help as visiting the states in a particular order does not make any difference. However, pruning in this case is helpful as it reduces the overall size of the state space. In addition to this, pruning is also useful in verifying protocols that are safe i.e., the specified security property is ”true” in all states.

5.1.1 Experiments for *Arbitrary Join*

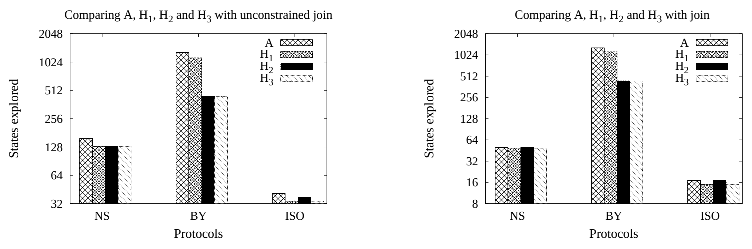
Hypothesis The first set of experiments is designed where contexts are freely formed, namely the join formula is just ”true”. In the absence of any constraints on the principals joining a session, the state space grows exponentially and this can possibly be controlled with the help of heuristics. We investigate how efficient are the heuristics in the absence of any constrained *join*.

Setting This set of experiments consists of eight protocols. All protocols are tested with arbitrary join and three instances of principals in each case. Furthermore, we have also tested a subset of these test cases (i.e., NS, ISO, and the BY protocols) with 2 instances of principals participating in each session.

Results Table 5.1 and Table 5.2 summarise the results for the experiments conducted to see the efficiency of heuristics in the absence of any join formula. The first two columns of each table respectively give the protocols that have been taken as test cases and presence of attacks in the protocol. The next two columns yield the results for pruning and re-ordering. In both cases we report the results for $\mathcal{A}$SPASyA and our three heuristics. An entry marked NA (after ”No Attack”) represents a fact that a protocol is free of any attack therefore the numbers for re-ordering will correspond to the numbers for pruning.

Table 5.1: Protocols with *arbitrary join* and 2 instances

Protocols	Attacks	Pruning				Re-ordering			
		$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$	$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$
NS	Yes	158	129	130	129	67	25	39	25
ISO	Yes	41	34	37	34	14	12	14	12
BY	Yes	1292	1134	440	438	335	873	179	177



(a) *Arbitrary join* (b) *Constrained join*

Figure 5.1: No. of states explored with *arbitrary* & *constrained join* for 2 instances

Table 5.2: Protocols with *arbitrary join* and 3 instances

Protocols	Attacks	Pruning				Re-ordering			
		$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$	$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$
NS	Yes	5183	4942	4943	4942	1288	197	1048	197
KSL	No	10240	3332	3607	3332	NA	NA	NA	NA
ISO	Yes	333	312	318	312	31	26	31	26
WMF	Yes	1055	563	838	563	451	111	196	111
Carlsen	No	9792	867	1142	867	NA	NA	NA	NA
DS	Yes	76273	31379	31381	31379	39438	22	26	22
BY	Yes	236497	226562	113512	113507	28995	132115	19065	9732
BKE	No	6316	5700	5701	5700	NA	NA	NA	NA

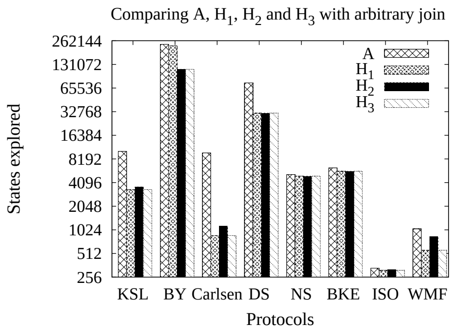


Figure 5.2: No. of states explored with *arbitrary join* for 3 instances

Interpretation Table 5.1 and Table 5.2 show that the overall efficiency of all heuristics is better than $\mathcal{A}$SPASyA when the state space is completely explored or when the search halts on the first attack. However, with the exception of the BY protocol, $\mathcal{H}_1$ is comparatively more efficient then $\mathcal{H}_2$ while their combination (i.e., $\mathcal{H}_3$) is no better than $\mathcal{H}_1$ (see § 5.1.4 for experiments demonstrating efficiency of $\mathcal{H}_2$).

The best results for pruning are obtained in the verification of DS, KSL, and Carlsen where more then 50% of the original state space is cut; more precisely, 58% in the case of DS, 67% in the case of KSL and 88% in the case of Carlsen. The least efficiency has been observed for the ISO and the NS where the reduction is roughly 4.5% of the original state space (Table 5.2). For re-ordering, the highest efficiency has been observed in the case of DS protocol and the least efficiency is in the case of ISO. Figure 5.1(a) and Figure 5.2 graphically represent the comparison between $\mathcal{A}$SPASyA and our heuristics for an *arbitrary join* formula.

5.1.2 Experiments for *Constrained Join*

Hypothesis This set of experiments aims to gauge the performance of heuristics in the presence of a *constrained join*. Recall that a *join* formula specifies the valid *connections* between the participants and thus restricts the state space by pruning those contexts that are not validated by the specified *join* formula (cf. § 2.6.2). Thus heuristics performance will not be as good as in the case of *arbitrary join*. We use these experiments to see if the heuristics can show any improvement over $\mathcal{A}$SPASyA in the presence of a *constrained join*.

Setting For the above hypothesis, we designed 10 different experiments that were carried on a total of seven protocols. All the seven protocols are tested with three instances of principals participating in a session and three of the protocols (i.e., NS, ISO, and BY) are also tested with (maximum of) two instances of principals participating in a session. In each case an appropriate *join* formula is also specified.

Results Table 5.3 and Table 5.4 highlight the results when a non-trivial join formula is defined. The former gives the results for protocols executed with (maximum of) two instances of the principal instances and the latter with (maximum of) three instances of the principal instances participating in each session.

Table 5.3: Protocols with *constrained join* and 2 instances

Protocols	Attacks	Pruning				Re-ordering			
		$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$	$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$
NS	Yes	50	49	50	49	39	38	39	38
ISO	No	17	15	17	15	NA	NA	NA	NA
BY	Yes	1292	1134	440	438	335	873	179	177

Table 5.4: Protocols with *constrained join* and 3 instances

Protocols	Attacks	Pruning				Re-ordering			
		$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$	$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$
NS	Yes	1589	1588	1589	1588	1048	599	1048	599
KSL	No	550	275	550	275	NA	NA	NA	NA
ISO	Yes	138	131	138	131	89	84	89	84
WMF	Yes	522	247	522	247	497	222	497	222
Carlsen	No	534	259	534	259	NA	NA	NA	NA
BY	Yes	236497	226562	113512	113507	28995	132115	19065	9732
Yahalom	Yes	2706	1559	2706	1559	1990	983	1990	983

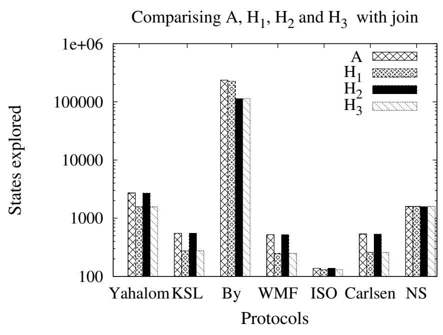


Figure 5.3: No. of states explored with *constrained join* for 3 instances

Interpretation It follows from the results that for both pruning and re-ordering heuristics $\mathcal{H}_1$ and $\mathcal{H}_3$ give a reduction in the state space but $\mathcal{H}_2$ does not show any improvement over $\mathcal{A}$SPASyA (with the exception of the BY protocol). This is due to the fact that the use of join formulae in $\mathcal{A}$SPASyA can avoid that some of the initial nodes are expanded; thus a join formula may be as effective as our heuristics. Nevertheless, the join formulae must be specified by the user and it is not simple to design them. Moreover, a weak join formula does not give results comparable to the cut produced by our heuristics. Figure 5.1(b) and Figure 5.3 are the graphical representation for the results of Table 5.3 and Table 5.4 respectively.

5.1.3 Experiments with Four Instances

Hypothesis The aim of these experiments is to show that heuristics are *effective* in addition to be efficient. More precisely, the size of the state space increases with an increase in the number of principal instances participating in a session and can possibly result into the state space explosion. We show with the help of the experiments that our heuristics may possibly avoid the state space explosion by reducing the size of the state space.

Setting Three protocols are taken as test cases, each of the protocol is executed with *arbitrary join* as well as a *constrained join*. The protocols are tested with four instances of the participants in each session.

Results Table 5.5 and 5.6 show the results for various protocols executed with 4 instances. The former gives the results for arbitrary join and the latter for constrained join. An entry in the table marked as "OF" suggests state space explosion.

Table 5.5: 4 instances and *arbitrary join*

Protocols	Attacks	Pruning				Re-ordering			
		$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$	$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$
NS	Yes	378265	375596	375597	375596	60192	6550	57524	6550
KSL	No	OF	2717158	OF	271758	NA	NA	NA	NA
Carlsen	No	OF	12140	OF	12140	NA	NA	NA	NA

Table 5.6: 4 instances and *constrained join*

Protocols	Attacks	pruning				Re-ordering			
		$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$	$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$
NS	Yes	92813	92812	92813	92812	60192	6550	57524	6550
KSL	No	OF	2717158	OF	271758	NA	NA	NA	NA
Carlsen	No	OF	12140	OF	12140	NA	NA	NA	NA

Interpretation As said, as number of instances increases, the state space grows and can possibly explode.

Table 5.5 shows that, for KSL and Carlsen verified with 4 instances, $\mathcal{A}$SPASyA gives an error and terminates as it can not handle the state space explosion. However, $\mathcal{H}$-ASPASyA (with $\mathcal{H}_1$ and $\mathcal{H}_3$) successfully verifies the protocols.

5.1.4 Experiments with $\mathcal{H}_2$

Hypothesis As evident from Tables 5.2 and 5.4, the efficiency shown by $\mathcal{H}_2$ is not comparable to $\mathcal{H}_1$ (except for few cases) due to the reasons illustrated in § 3.3.2. We want to show with the help of these experiments that the heuristic $\mathcal{H}_2$ can outperform

$\mathcal{H}_1$ in verifying some special cases. More precisely, the performance of $\mathcal{H}_2$ is best with a security property that specifies symmetric keys in addition to principal identities.

Setting We have designed experiments to specify the *conditions* under which $\mathcal{H}_2$ can outperform $\mathcal{H}_1$. Three *symmetric key* protocols ISO, WMF, and Yahalom have been chosen as test cases. All protocols have been tested with *arbitrary join* and three instances of the principal instances to check the secrecy of private keys and proper sharing of private keys.

Results Table 5.7 shows the results from the experiments conducted to demonstrate the efficiency of $\mathcal{H}_2$.

Table 5.7: 3 instances and *arbitrary join*

Protocols	Attacks	Pruning				Re-ordering			
		$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$	$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$
ISO	yes	315	295	265	260	31	26	31	26
WMF	no	406	453	391	453	1104	999	978	999
Yahalom	no	1750	1690	1742	1682	2869	2809	2768	2708

Interpretaion As expected, $\mathcal{H}_2$ performs better than $\mathcal{H}_1$ as can be observed from the Table 5.7. In particular, $\mathcal{H}_2$ performs better than $\mathcal{H}_1$ for pruning in the case of ISO and Yahalom protocols and for the re-ordering in the case of WMF protocol.

The reason for the efficiency of $\mathcal{H}_2$ is the security formula. For instance, consider the ISO protocol presented in the § 3.3.2. The Figure 3.4(a) represents the state space generated for the ISO protocol for which actual number of states explored is given in Table 5.7. Note that $\mathcal{H}_1$ prunes the state with the context $\{B_1, B_2, B_3\}$. However, heuristic $\mathcal{H}_2$ can utilise the information given in the security formula (about symmetric keys) to prune some additional traces. This is reflected in the pruning results of ISO given in Table 5.7, where $\mathcal{H}_2$ explores 265 sates compared to 295 states explored by $\mathcal{H}_1$ making a difference of 30 states in overall size of the state space.

Noteworthy, the first attack is discovered rather quickly by $\mathcal{H}_1$ as compared to $\mathcal{H}_2$. Apparently, the re-ordering capability of $\mathcal{H}_1$ is better than $\mathcal{H}_2$ as $\mathcal{H}_1$ discovers attack in fewer states. However, analysis of the results show that the attack is in the first trace emerging from the context $\{A_1, A_2, A_3\}$ (c.f., Figure 3.4(a)) which is by default the first state to be explored by $\mathcal{A}$SPASyA. This suggests that results for $\mathcal{A}$SPASyA and $\mathcal{H}$-ASPASyA should be same. In fact, the efficiency shown by $\mathcal{H}_1$ in discovering first attack is actually due to pruning rather on to re-ordering. The context $\{B_1, B_2, B_3\}$ was not explored further by $\mathcal{H}_1$ thereby discovering the first attack quicker.

5.2 Time Analysis

Hypothesis The aim of this set of experiments is to see the efficiency achieved in terms of time when exploring the state space with the help of the heuristics $\mathcal{H}_1$,$\mathcal{H}_2$, and $\mathcal{H}_3$.

Settings Five protocols have been taken as test cases. All the protocols have been tested with arbitrary *join* as well as constrained *join*.

Results Tables 5.8 and 5.9 report the execution time [2] for various protocols with respect to *arbitrary join* and *constrained join* respectively.

[2]The time reported is the aggregate of the time spent by the process in user and kernel mode and corresponds to the execution on a Pentium 4 CPU 3.20 GHz with 1010.3 MB of memory

Table 5.8: Execution time for protocols using *arbitrary join*

Protocols	Instances	Time (mm:ss:ms)			
		$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$
NS	2	00:01:780	00:01:824	00:01:888	00:01:788
NS	3	00:17:501	00:16:141	00:16:937	00:16:224
ISO	2	00:01:628	00:04:436	00:01:652	00:01:644
ISO	3	00:04:564	00:01:684	00:04:597	00:04:508
KSL	2	305:37:358	02:35:466	299:06:809	03:34:741
BKE	2	00:02:440	00:02:364	00:02:420	00:02:504
BKE	3	00:12:953	00:12:117	00:12:565	00:12:212
DS	3	09:11:446	00:21:759	14:12:737	08:42:676

Table 5.9: Execution time for protocols using *constrained join*

Protocols	Instances	Time (mm:ss:ms)			
		$\mathcal{A}$	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$
NS	2	00:01:664	00:01:806	00:01:764	00:01:704
NS	3	00:05:412	00:05:240	00:05:500	00:05:284
NS	4	06:46:449	06:50:538	07:17:611	06:29:964
ISO	2	00:01:500	00:01:596	00:01:532	00:01:484
ISO	3	00:02:028	00:02:096	00:02:020	00:02:044
KSL	3	04:07:299	00:14:901	00:01:520	00:12:077
DS	3	00:04:748	00:02:632	00:04:756	00:02:728

Interpretation We get efficiency in terms of time for most of the protocols which is most significant in case of KSL where the verification took almost 5 hours for $\mathcal{A}$SPASyA as compared to 2 min by $\mathcal{H}_1$ (see Table 5.8). Note that the KSL was also the best protocol when testing our heuristics in terms of the state space. For the protocols such as NS, the gain is not much significant (the time savings is in seconds); this suggests that efficiency of heuristics in terms of time savings is not directly propotional to the efficiency in terms of the state space reduction.

Remark As shown in [43], $\mathcal{A}$SPASyA's benchmark are comparable to other state-of-the-art verification frameworks.

5.3 Analysis of the Results

We report the following observations.

- $\mathcal{A}$SPASyA and $\mathcal{H}$-ASPASyA yield the same attacks. This is not surprising for $\mathcal{H}_1$ because its correctness has been proved in [73]. The correctness of $\mathcal{H}_2$ and $\mathcal{H}_3$ has not been demonstrated, however our experiments hint that they are correct. We remark that our DMC approach is effective as $\mathcal{H}_1$ and $\mathcal{H}_2$ cut the state space. Hence, as illustrated in Table 5.2, the cases with no attack are in general more efficiently explored in $\mathcal{H}$-ASPASyA.

- As proved by Table 5.5, DMC mitigates the state explosion problem to the point that memory overflow is avoided in a few cases. Notice that for small state spaces DMC yields a limited improvement. For instance, in the case of NS executed with three principal instances there is a reduction of 4.4 % of the total state space (cf., Table 5.2). On the other hand, as the state space grows DMC may drastically reduce the search space. For instance, Table 5.2 shows that in the Carlsen protocol with three instances only 9% of the state space is explored by $\mathcal{H}$-ASPASyA.

- Our heuristics play an important role in the absence of any *constrained join*. It can be observed from Table 5.2 that there is a huge reduction in the state space in the case of Carlsen and KSL protocols. For the KSL protocol without a join formula we are able to cut-down almost 67% of the original state space.

- The results for re-ordering are encouraging for heuristic $\mathcal{H}_1$ and $\mathcal{H}_3$. For instance, in the case of the DS protocol the first attack is discovered by exploring a very tiny portion of the state space by $\mathcal{H}$-ASPASyA while $\mathcal{A}$SPASyA visits a large number of states. However, the results of heuristic $\mathcal{H}_2$ for re-ordering are not as significant as for pruning. The re-ordering capability of $\mathcal{H}_2$ can be improved as mentioned in § 3.3.1.

5.4 Choosing the Right Heuristic

We remark that heuristics are ”rules of thumb” and to describe precise conditions under which each heuristic can give best performance is hard. However, we provide a basic comparison and a few suggestions that can serve as guidelines for using appropriate heuristics according to the choice of the protocol and security property in question.

As observed from results, the overall efficiency of $\mathcal{H}_1$ is better than the efficiency of $\mathcal{H}_2$. $\mathcal{H}_1$ is a better choice when the verifier is interested in finding an attack on a protocol. However, when the security property of interest refers to symmetric keys, $\mathcal{H}_2$ seems to be more reasonable choice. The behaviour of $\mathcal{H}_3$ for re-ordering is difficult to predict. Usually, the efficiency of $\mathcal{H}_3$ is equal to $\mathcal{H}_1$ except for few cases where $\mathcal{H}_2$ can outperform $\mathcal{H}_1$ (such as WMF verification, cf., Table 5.7).

For complete state space exploration, we consider $\mathcal{H}_3$ to be best choice regardless the nature of the protocol and the property to verify. We argue that contrary to re-ordering, $\mathcal{H}_3$ is the best heuristic. The reason is that pruned states can never be recovered and therefore $\mathcal{H}_2$ can only add to the set of states pruned by $\mathcal{H}_1$. Therefore, the efficiency of $\mathcal{H}_3$ is either equal to or better than the individual efficiencies of $\mathcal{H}_1$ or $\mathcal{H}_2$ (whichever is maximum).

5.5 On the Experimental Setting

In this section we discuss a few aspects of our experimental settings. More precisely, we investigate the impact of chosen test cases, verification tool, and properties. Moreover, we discuss if our experimental setting for verification can possibly affect the efficiency of our heuristics and consequently the generality of our approach.

5.5.1 Choosing the Verification Framework and the Tool

The verification framework chosen to define the heuristics is the one given in [21]. However, as mentioned in § 1.1, the idea of our heuristics can be adapted by other security verification frameworks. The framework in [21] is taken as the test case to demonstrate that the heuristics can possibly enhance the performance of the MC algorithm by quickly exploring the attacks and by possibly reducing the size of the state space.

The framework was chosen due its different features discussed in § 2.6.1.2. Here, we only summarise the key characteristics of the framework.

The framework provides a convenient way to express the security properties and to formalise the protocols. These two aspects are independent of each other as against some verification frameworks where security property is embedded in the formalised protocol [74]. This simplifies the verification task as verifying the protocol for different properties will require the alteration of security property alone and not the formalised protocol.

Most of the attacks on security protocols are due to the capability of the intruder to collect information from different sessions and manipulate it. $\mathcal{A}$SPASyA supports multi-session analysis, by uniquely identifying each session, its participants, and the variables associated with each participant.

The framework also supports a wide range of security and authentication properties that can be expressed in $\mathcal{PL}$. Furthermore, it provides the unique feature of *join* formula which allows the user to specify the interested scenarios for verification.

5.5.2 Choosing Security Protocols

In this section, the factor under consideration is the set of security protocols that are taken as test cases for evaluating the efficiency of our heuristics. More specifically, one could argue that the security protocols we have considered are indeed the ones that highlight the efficiency of our designed heuristics. In this context we report the following observations.

The heuristics $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ are property-specific, i.e., they rely on the property specification to derive their hints in order to evaluate the states. This suggests that the heuristics are in fact protocol independent and their efficiency does not rely on the choice of the protocol.

However, we would also like to mention here that heuristics rely on domain specific hints to evaluate the states. Therefore, it is certainly possible that the heuristics behave differently with different types of the the protocols. For example, as mentioned in § 3.3.2, heuristic $\mathcal{H}_2$ performs better for symmetric key protocols as against heuristics $\mathcal{H}_1$ and $\mathcal{H}_3$. Having said that, we emphasize that actual efficiency of the heuristic $\mathcal{H}_2$ depends on the security property rather than on the protocol. This amounts to say that a symmetric key protocol alone would not be sufficient for $\mathcal{H}_2$ to produce better efficiency. It will only perform better when the security property also specifies symmetric keys in the property specification.

5.5.3 Choosing Security Properties

As mentioned earlier in this section, our heuristics are property-based. Hence, security properties are the main factor that can affect the efficiency of the heuristics. In the following we argue that our results are not biased by the choice of the properties.

Most of our protocols are tested for secrecy, integrity, and authentication. These are rather general properties which are considered in most of the works done in the area of security protocol verification.

Heuristic $\mathcal{H}_1$ exploits the quantification over principal instances and checks if such principals have joined the context. Furthermore, if the security property specifies universal quantification over a principal instance which never joins the context, then the context can be safely pruned. Therefore, we conjecture that the pruning capability of $\mathcal{H}_1$ increases with the number of universal quantifiers in the security property. Note that our test cases are not biased on the security property. In out of nine test cases, only two test cases specify more than one universal quantification in the security property. This shows that the test cases are devised without keeping in mind the specific features that enhance the performance of the heuristics.

For heuristic $\mathcal{H}_2$ we think that it yields better efficiency if security property specifies the symmetric keys as in the case of ISO example illustrated in § 3.3.2. Again, note that the security protocols and their properties to be verified are chosen irrespective of the features required by the heuristics to perform strongly. We have conducted only two experiments where security property specifies symmetric key.

Finally, we would like to add that if the protocol is attack free, re-ordering capability has no role to play. In such cases, our heuristics have pruning capability that can contribute towards efficient exploration of the state space. Further note that we have included a substantial number of such experiments where protocol enjoys the property. More precisely, there are twelve experiments out of thirty that resulted into the outcome ”no attack” for the protocol under specified property.

Chapter 6

Summary and Future Directions

6.1 Published Works

The heuristic $\mathcal{H}_1$ was first time presented in [73]. The paper proposed the idea of using heuristics for security protocol verification and contained the definition of $\mathcal{H}_1$ along with proof of its correctness. Our paper [72] presented formal verification of a *federated identity management protocol* using $\mathcal{A}$SPASyA. This thesis takes a few examples from [72], the definition of $\mathcal{H}_1$ from [73], and the proof for the correctness of $\mathcal{H}_1$ from [73]. The thesis further extends the works presented in [72] and [73] by implementing $\mathcal{H}_1$, defining two new heuristics, implementing $\mathcal{H}_2$ and $\mathcal{H}_3$, and performing an analysis of the experimental results.

6.2 Summary

The idea of incorporating intelligent search strategies in model checking to increase efficiency is not novel. Model checking with heuristic search strategies is commonly known as directed model checking [54] and a few significant works employing directed model checking are [40, 46, 39, 53]. Surprisingly, though directed model checking has found its way in general model checkers, its feasibility for security protocol verification has not been investigated vigorously. To the best of our knowledge, the only work that mentions the use of heuristics in security protocol verification is [34, 11].

In this context, we studied some heuristics tailored for security protocol verification. The study was carried out using the formalism given in [43], which comprises of a process calculus to encode protocols, a logic to define security properties, and a tool called $\mathcal{A}$SPASyA. We have defined three heuristics that are property-dependent as they direct the search towards the states that violate the security property; more precisely, they are property-specific as they rely on the property to be checked to weight the states. An additional feature of our heuristics is pruning, namely the states that cannot lead to an attack are curtailed. The heuristics therefore have a dual role to play (i) to explore the states according to their likeliness of leading to an attack and (ii) pruning the states that have no chance of leading to an attack. We consider our DMC approach to be different from others as (i) we use heuristic functions that are evaluation based in contrast to other works such as [40, 46, 53] where an estimation based function is used (ii) only a part of the state space is explored by means of directed search.

The first heuristic (called $\mathcal{H}_1$) exploits security formulae and weights join transitions. Each context is heuristically evaluated by analysing its principal instances and the way they are quantified in security formulae. A principal that is universally quantified in the security formula but is absent from a context suggest a possible violation of the security property. The second heuristic weights states by looking at open variables in the security formula. The states that violate the assignments of open variables (as specified in security formula) are explored before the others. The third heuristic $\mathcal{H}_3$ is composite i.e., it can rank a state either with $\mathcal{H}_1$ or $\mathcal{H}_2$ whichever is appropriate for the state in question.

The heuristics are incorporated into the symbolic model checker $\mathcal{A}$SPASyA yielding a new tool $\mathcal{H}$-ASPASyA. A number of protocols have been tested with $\mathcal{H}$-ASPASyA, with each protocol tested with $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ to analyse the respective efficiency of each heuristic. The overall results are encouraging and demonstrate effectiveness of our approach. In some cases the heuristics explore only a fraction of the state space to discover attacks. Although, all three heuristics contribute to efficiency, generally $\mathcal{H}_3$ outperforms the other two heuristics. Along with contributing to efficiency, heuristics are able to successfully verify a few protocols that could not be verified with conventional model checking approach due to state space explosion.

Heuristic $\mathcal{H}_1$ has been proved correct, namely pruned parts of the state space are free of attacks. For $\mathcal{H}_2$ and $\mathcal{H}_3$ there are no such formal results but experiments suggest their correctness as $\mathcal{H}$-ASPASyA (with $\mathcal{H}_2$ and $\mathcal{H}_3$) reports all those attacks that were initially exposed by $\mathcal{A}$SPASyA.

6.3 Future Directions

The model checking algorithm for $\mathcal{H}$-ASPASyA is a hybrid of directed and conventional model checking. More precisely, our heuristics are applied only in an initial phase of the verification where the contexts containing the instances of participants are formed. In this phase, the contexts are cut and ordered according to our heuristics and then, in a second phase, they are model-checked without further intelligent search. This hints that further improvements could be obtained by exploiting more heuristics. For instance, it would be straightforward to define a heuristic for communication transitions working on the idea of $\mathcal{H}_2$. Further, one could also define general heuristics about the choice of messages the attacker sends to the protocol so to further limit the size of the state space.

Although our heuristics are tailored for the framework introduced in [43], they can be easily adapted to other security verification frameworks. Since every model checking framework has a notion of security property similar to $\mathcal{H}$-ASPASyA therefore they can use our idea of heuristics to benefit from the directed model checking approach.

We also plan to compare $\mathcal{H}$-ASPASyA with other state-of-the-art model checkers. Note that typically the efficiency of model checkers is not compared quantitatively (i.e., on the basis of time or space) such as [12, 53]. The comparison with other tools is either done on the basis of model checking approach or on the restrictions on the execution model of the protocol. A quantitative comparison as noticed in [33] is hard to perform due to a number of reasons. Mainly, the infinite state spaces are differently dealt with by each tool; more precisely, each tool imposes some kind of restrictions on the behaviours of the protocol explored. These restrictions determine the size of the state space to be explored. Thus each tool essentially explores a few possible behaviours and gives its results according to those behaviours. This makes it difficult to compare two different tools. It is suggested in [34] to compare the tools by first generating the uniform state spaces for each tool and then testing the protocols on these comparable state spaces to see the relative efficiency of the tools. This demands ample amount of time as it requires a protocol to be modelled in different formalisms, determining the comparable state spaces in different tools, and expert knowledge of tools under the analysis. Due to shortage of time, quantitative comparison of $\mathcal{H}$-ASPASyA with other tools has not been performed and remains in the scope of future work.

Appendix A

Security Protocols

1. NS, Needham Schroeder Public Key Protocol [25], [61]

Formalized Protocol

$$\begin{array}{lll} A: & (x) & [out(\{(A,na)\}_{x^+}).in(\{(na,?y)\}_{A^-}).out(\{y\}_{x^+})] \\ B: & () & [in(\{(?z,?u)\}_{B^-}).out(\{(u,nb)\}_{z^+}).in(\{nb\}_{B^-})] \end{array}$$

Security Property

$$\phi_{NS} = (\forall i : B.\ (\kappa \triangleright nb_i \implies z_i = I_0) \vee (\exists j : A.\ z_i = A_j \implies x_j = B_i))$$

2. WMF, Wide Mouthed Frog Protocol [25]

Formalized Protocol

$$\begin{array}{lll} A: & (x,s) & [out((A,\{na,(x,kab)\}_s))] \\ S: & (u,a,v,b) & [in((u,\{(?t,(v,?r))\}_a)).out(\{(ns,(u,r))\}_b)] \\ B: & (s) & [in(\{(?t,(?z,?r))\}_s)] \end{array}$$

Security Property

$$(\forall j : B.\ \exists l : S.\ \exists i : A.$$
$$((v_l = B_j \wedge u_l = A_i \wedge x_i = B_j) \implies (t_l = na_i \wedge t_j = ns_l \wedge r_j = kab_i)))$$

3. KSL, Kehn Schönwälder Langendörfer Protocol [51]

Formalised protocol

$$\begin{array}{lll} A: & (b,sk) & [out(na,A).in((\{((na,b),?r)\}_{sk},?tkb),?bn),\{na\}_r)).out(\{bn\}_r)] \\ B: & (sk) & [in((?cn,?u)).out((((cn,u),nb),B)).in((\{((nb,u),?r)\}_{sk},?tka)). \\ & & out((((tka,\{((nt,u),r)\}_{kbb}),nc),\{cn\}_r)).in(\{nc\}_r)] \\ S: & (a,ak,b,bk) & [in((((?cna,a),?cnb),b)).out((\{((cnb,a),kab)\}_{bk},\{((cna,b),kab)\}_{ak}))] \end{array}$$

Security Property

$$\begin{aligned} \phi_{ksl} = (\forall l : B.\ \ & \exists\, i : S. \exists j : A. \\ & (b_i = B_l \wedge b_j = B_l \wedge a_i = A_j) \implies cn_l = na_j \\ & \wedge\ u_l = A_j \wedge\ bn_j = nc_l \wedge\ r_j = kab_i \\ & \wedge\ r_l = kab_i \wedge\ cna_i = na_j \wedge\ cnb_i = nb_l) \end{aligned}$$

4. ISO, ISO/IEC Protocol [50]

Formalized Protocol

$$\begin{array}{lll} A: & (b,sk) & [in((?r,?td)).out((na,\{((r,b),nc)\}_{sk}))] \\ B: & (sk) & [out((nb,nd)).in((?ta,\{((nb,B),?tc)\}_{sk}))] \end{array}$$

Security Property

$$\forall i : A.\ \exists j : B.\ b_i = B_j \implies r_i = nb_j \wedge tc_j = nc_i$$

5. Carlsen, Carlsen Protocol [26]

Formalized protocol

$$\begin{array}{lll}
A: & (b,sk) & [out((A,na)).in(((\{((na,b),?r)\}_{sk},\{na\}_{r}),?tb)).out(\{tb\}_{r})] \\
B: & (sk) & [in((?a,?ta)).out((((a,ta),B),nb)).in((\{((?r,nb),a)\}_{sk},?c)). \\
 & & out(((c,\{ta\}_{r}),nc)).in(\{nb\}_{r})] \\
S: & (u,uk,v,vk) & [in((((u,?tu),v),?tv)).out((\{((kab,tv),u)\}_{vk},\{((tu,v),kab)\}_{uk}))]
\end{array}$$

Security Property

$$\forall i:A.\ \ \exists j:B.\exists l:S.\ u_l = A_i \wedge v_l = B_j \wedge b_i = B_j \implies$$
$$a_j = A_i \wedge r_i = r_j \wedge r_j = kab_l \wedge tb_i = nb_j \wedge ta_j = na_i \wedge tu_l = na_i \wedge tv_l = nb_j$$

6. DS, Denning-Sacco Protocol [35]

Formalized protocol

$$\begin{array}{lll}
S: & (\,) & [in((?x,?y)).out((\{(x,x^{+})\}_{S^{-}},\{(y,y+)\}_{S^{-}}))] \\
A: & (x,y) & [out((A,y)).in((\{(A,A^{+})\}_{x^{+}},\{(y,?z)\}_{x^{+}})). \\
 & & out(((\{(A,A^{+})\}_{x^{-}},\{(y,z)\}_{x^{-}}),\{k\}_{A^{-}}\}_{z}))] \\
B: & (x) & [in(((\{(?w,?y)\}_{x^{+}},(B,B^{+})_{x^{+}}),\{\{?r\}_{y}\}_{B^{-}}))]
\end{array}$$

Security Property

$$\forall i:B.\ \kappa \not\triangleright r_i \wedge \forall j:B.\ \exists k:A.\ \ y_j = A_k \implies y_k = B_j$$

7. BY, Beller-Yacobi Protocol [13]

Formalized protocol

$$\begin{aligned} A: \quad (b,s) \quad & [out((A,A+)).in(\{?r\}_{A-}).out(\{na\}_r). \\ & in(\{(((b,?bk),\{(b,bk)\}_s),\{na\}_{b^-}\}_r)] \\ B: \quad (s) \quad & [in((?a,?ak)).out(\{kab\}_{ak}).in(\{?ta\}_{kab}). \\ & out(\{(((B,B+),\{(B,B+)\}_s),\{ta\}_{B^-})\}_{kab})]; \end{aligned}$$

Security Property

$$\begin{aligned} \forall i:B. \quad & (\kappa \rhd kab_i \implies a_i = I_0) \vee (\exists j:A.b_j = B_i \implies \\ & r_j = kab_i \wedge bk_j = B_i^+ \wedge ta_i = na_j \wedge \kappa \not\rhd kab_i)) \\ & \vee(\forall\, l:A.(\kappa \rhd na_l \implies b_l = I_0)) \end{aligned}$$

8. BKE, Bilateral Key Exchange Protocol [29]

Formalized protocol

$$\begin{aligned} A: \quad () \quad & [in((?b,\{(?tb,b)\}_{A^-})).out(\{(((tb,na),A),kab)\}_{b^+}).in(\{na\}_r)] \\ B: \quad (a) \quad & [out((B,\{(nb,B)\}_{a^+})).in(\{(((nb,?ta),a),?r)\}_{B^-}).out(\{ta\}_r)]; \end{aligned}$$

Security Property

$$\begin{aligned} \forall i:B.(\exists j:A. \quad & a_i = A_j \implies tb_j = nb_i \wedge ta_i = na_j \\ & \wedge r_i = kab_j \wedge b_j = B_i \wedge \kappa \not\rhd r_i) \\ & \vee(a_i = I_0) \end{aligned}$$

9. Yahalom, Yahalom Protocol [25]

Formalized protocol

$$\begin{array}{lll} A: & (b,sk) & [out((A,na)).in((\{(((b,?r),na),?tb)\}_{sk},?c)).out((c,\{tb\}_r))] \\ B: & (sk) & [in((?a,?ta)).out((B,\{((a,ta),nb)\}_{sk})).in((\{(a,?r)\}_s k,\{nb\}_r))] \\ S: & (u,uk,v,vk) & [in((u,\{((v,?tv),?tu)\}_{uk})).out((\{(((u,kab),tv),tu)\}_{vk},\{(v,kab)\}_{uk}))]; \end{array}$$

Security Property

$$\begin{array}{ll} \forall j:A.\ \exists l:B.\exists i:S. & u_i = B_l \wedge v_i = A_j \wedge b_j = B_l \implies \\ & r_j = r_l \wedge r_l = kab_i \wedge ta_l = tv_i \wedge tv_i = na_j \wedge tu_i = tb_j \wedge tb_j = nb_l \end{array}$$

Bibliography

[1] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, January 1999.

[2] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in Model Checking. In *Proceedings of 10th Internationl Conference on Computer Aided Verification*, pages 521–525, 1998.

[3] Rajeev Alur and Bow-Yaw Wang. ”Next” Heuristic for On-the-Fly Model Checking. In *CONCUR ’99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 98–113. Springer-Verlag, London, UK, 1999.

[4] Roberto M. Amadio, Denis Lugiez, and Vincent Vanackère. On the symbolic reduction of processes with cryptographic functions. *Theoretical Computer Science*, 290(1):695–740, 2003.

[5] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: An Interactive and Automatic Tool for Proving Theorems of Type Theory. In *Proceedings of 6th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 366–370, 1993.

[6] Peter B. Andrews, Chad E. Brown, Frank Pfenning, Matthew Bishop, Sunil Issar, and Hongwei Xi. ETPS: A System to Help Students Write Formal Proofs. *Journal of Automated Reasoning*, 32(1):75–92, 2004.

[7] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Proceedings of 17th Internationl Conference on Computer Aided Verification*, pages 281–285, 2005.

[8] Alessandro Armando and Luca Compagna. SAT-based Model-Checking for Security Protocols analysis. *International Journal of Information Security*, 7(1): 3–32, 2008.

[9] Giacomo Baldi. Security Protocols Verification by Means of Symbolic Model Checking. `http://www.di.unipi.it/~etuosto/aspasya/aspasya.html`, 2003. [Online; accessed 19-Oct-2011].

[10] Giacomo Baldi, Andrea Bracciali, Gianluigi Ferrari, and Emilio Tuosto. A Coordination-based Methodology for Security Protocol Verification. In Nadia Busi, Roberto Gorrieri, and Fabio Martinelli, editors, *International Workshop on Security Issues with Petri Nets and other Computational Models*, volume 121 of *Electronic Notes in Theoretical Computer Science*, pages 23–46. Elsevier, Bologna (Italy), June 26 2004, February 2005.

[11] David A. Basin. Lazy Infinite-State Analysis of Security Protocols. In *Proceedings of the International Exhibition and Congress on Secure Networking - CQRE (Secure) '99*, pages 30–42. Springer-Verlag, London, UK, 1999.

[12] David A. Basin, Sebastian Mödersheim, and Luca Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In *Proceedings of Eighth European Symposium on Research in Computer Security '03*, LNCS 2808, pages 253–270. Springer-Verlag, Heidelberg, 2003.

[13] M.J. Beller and Y. Yacobi. Fully-fledged Two-way Public Key Authentication and Key Agreement for Low-cost Terminals. *Electronics Letters*, 29, 1993.

[14] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, CSFW '01, pages 82–. IEEE Computer Society, Washington, DC, USA, 2001.

[15] Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Symbolic Guided Search for CTL Model Checking. In *Conference on Design Automation (DAC)*, pages 29–34, 2000.

[16] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. *Lecture Notes in Computer Science*, 2127:27–34, 2001.

[17] Michele Boreale. Symbolic trace analysis of cryptographic protocols. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Colloquium on Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2001.

[18] Michele Boreale and Maria Grazia Buscemi. A Method for Symbolic Analysis of Security Protocols. *Theoretical Computer Science*, 338, Issues 1-3:393–425, 2005.

[19] Johannes Borgström, Sébastien Briais, and Uwe Nestmann. Symbolic Bisimulation in the Spi Calculus. In Philippa Gardner and Nobuko Yoshida, editors, *International Conference in Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 161–176. Springer-Verlag, 2004.

[20] Liana Bozga, Yassine Lakhnech, and Michaël Périn. HERMES: An Automatic Tool for Verification of Secrecy in Security Protocols. In *Proceedings of 15th Internationl Conference on Computer Aided Verification*, pages 219–222, 2003.

[21] Andrea Bracciali, Gianluigi Ferrari, and Emilio Tuosto. A Symbolic Framework for Multi-faceted Security Protocol Analysis. *International Journal of Information Security*, 7(1):55–84, 2008.

[22] Stephen H. Brackin. A HOL extension of GNY for Automatically Analyzing Cryptographic Protocols. In *Proceedings of 9th Computer Security Foundations Symposium*, pages 62–, 1996.

[23] N. Brownlee. *Formal Systems (Europe) Ltd. Failures-Divergence Refinement. FDR2 User Manual.* 2000.

[24] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10E20 States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[25] Michael Burrows, Martín Abadi, and Roger Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8:18–36, 1990.

[26] Ulf Carlsen. Optimal Privacy and Authentication on a Portable Communications System. *SIGOPS Operating System Review*, 28, 1994.

[27] Yannick Chevalier, Ralf Küsters, Michaël Rusinowitch, and Mathieu Turuani. An NP Decision Procedure for Protocol Insecurity with XOR. In *Annual Sym-*

posium on Logic in Computer Science, pages 261–270. IEEE Computer Society, 2003.

[28] Yannick Chevalier, Ralf Küsters, Michaël Rusinowitch, and Mathieu Turuani. Deciding the Security of Protocols with Diffie-Hellman Exponentiation and Products in Exponents. In Paritosh K. Pandya and Jaikumar Radhakrishnan, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *Lecture Notes in Computer Science*, pages 124–135. Springer-Verlag, 2003.

[29] John Clark and Jeremy Jacob. A survey of authentication protocol literature: Version 1.0. `http://www.cs.fsu.edu/yasinsac/group/work/childs/`, 1997.

[30] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. *Formal Methods of System Design*, 9:77–104, August 1996.

[31] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programamming Languages and Systems*, 16:1512–1542, September 1994.

[32] Edmund M. Clarke, Somesh Jha, and Wilfredo R. Marrero. Partial Order Reductions for Security Protocol Verification. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, (TACAS '00), pages 503–518. Springer-Verlag, 2000.

[33] Cas J.F. Cremers. Unbounded Verification, Falsification, and Characterization of Security Protocols by Pattern Refinement. In *Proceedings of the 15th ACM conference on Computer and Communications security*, pages 119–128. ACM, New York, NY, USA, 2008.

[34] C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006.

[35] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in Key Distribution Protocols. *Communications of the ACM*, 24, 1981.

[36] David L. Dill and C. Han Yang. Validation with Guided Search of the State Space. *Design Automation Conference*, 0:599–604, 1998.

[37] D. Dolev and A. Yao. On the Security of Public Key Protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, Mar 1983.

[38] Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Undecidability of Bounded Security Protocols, August 10 1999. URL `http://citeseer.ist.psu.edu/223423.html;ftp://www.cis.upenn.edu/pub/papers/scedrov/fmsp99.ps.gz`.

[39] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Protocol Verification with Heuristic Search. In *AAAI Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.

[40] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed Explicit-state Model Checking in the Validation of Communication Protocols. *International Journal of Software Tools and Technologies Transfer*, 5(2):247–267, 2004.

[41] Stefan Edelkamp, Viktor Schuppan, Dragan Bošnački, Anton Wijs, Ansgar Fehnker, and Husain Aljazzar. Model Checking and Artificial Intelligence. chapter Survey on Directed Model Checking, pages 65–89. Springer-Verlag, Berlin, Heidelberg, 2009.

[42] E. Allen Emerson, A. Prasad Sistla, and Hermann Weyl. Symmetry and Model Checking, 1994.

[43] Gianluigi Ferrari, Andrea Bracciali, and Emilio Tuosto. A symbolic framework for multi-faceted security protocol analysis. *Internation Journal of Information Security*, 7(1):55–84, 2008.

[44] W. Fokkink, M.T. Dashti, and A. Wijs. Partial Order Reduction for Branching Security Protocols. In *Proceedings of 10th International Conference on Application of Concurrency to System Design (ACSD)*, pages 191 –200, june 2010.

[45] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning about Belief in Cryptographic Protocols. In *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society Press, 1990.

[46] Sara Gradara, Antonella Santone, and Maria Luisa Villani. Formal Verification of Concurrent Systems via Directed Model Checking. *Electronic Notes in Theoretical Computer Science*, 185:93–105, 2007.

[47] Alex Groce and Willem Visser. Model checking Java programs Using Structural Heuristics. *SIGSOFT Softw. Eng. Notes*, 27:12–21, July 2002.

[48] J Hajek. Automatically Verified Data Transfer Protocols. In *Proceedings of the 4th International Compuetr Communications Conference*, 1978.

[49] Gerard Holzmann. *The Spin Model Checker, Primer and Reference manual.* Addison-Wesley Professional, first edition, 2003.

[50] ISO/IEC 9798. Information Technology - Security Techniques - Entity Authentication Mechanism Part 2: Entity Authentication Using Symmetric Techniques. Technical report, 1993.

[51] A. Kehne, J. Schönwälder, and H. Langendörfer. A Nonce-based Protocol for Multiple Authentications. *SIGOPS Operating System Review*, 26, October 1992.

[52] Axel Kehne, Jürgen Schönwälder, and Horst Langendörfer. Multiple Authentications with a Nonce-Based Protocol Using Generalized Timestamps. In *Proceedings of 11th International Conference on Computer Communications*. Genua, 1992.

[53] Sebastian Kupferschmid, Klaus Dräger, Jörg Hoffmann, Bernd Finkbeiner, Henning Dierks, Andreas Podelski, and Gerd Behrmann. Uppaal/DMC- Abstraction-Based Heuristics for Directed Model Checking. In *Proceedings of the Thirteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 679–682, 2007.

[54] Alberto Lluch Lafuente. *Directed Search for the Verification of Communication Protocols*. PhD thesis, June 2003.

[55] Gavin Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 56:131–133, November 1995.

[56] Gavin Lowe. An Attack on the Needham-Schroeder Public-key Authentication Protocol. *Information Processing Letters*, 56:131–133, November 1995.

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

[58] Catherine Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programramming*, 26(2):113–131, 1996.

[59] Robin Milner. *Communication and Concurrency*. Printice Hall, 1989.

[60] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols using Mur/spl phi/. In *Proceedings of the 18th IEEE Symposium on*

Security and Privacy, SP ’97. IEEE Computer Society, Washington, DC, USA, 1997.

[61] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of ACM*, 21:993–999, December 1978.

[62] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, NY, June 1992.

[63] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[64] Lawrence C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, January 1998.

[65] Frank Reffel and Stefan Edelkamp. Error detection with directed symbolic model checking. In *World Congress on Formal Methods*, pages 195–211, 1999.

[66] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[67] Vitaly Shmatikov. Decidable Analysis of Cryptographic Protocols with Products and Modular Exponentiation. In David Schmidt, editor, *European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 355–369. Springer-Verlag, April 2004.

[68] Dawn Xiaodong Song. Athena: a New Efficient Automatic Checker for Security Protocol Analysis. In *Proceedings of the 12th IEEE workshop on Computer Se-*

curity Foundations, pages 192–202. IEEE Computer Society, Washington, DC, USA, 1999.

[69] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand Spaces: Why is a Security Protocol Correct? In *IEEE Symposium on Security and Privacy*, pages 160–171, 1998.

[70] Emilio Tuosto. *Non-Functional Aspects of Wide Area Network Programming*. PhD thesis, Dipartimento di Informatica, Università di Pisa, May 2003. TD-8/03.

[71] Mathieu Turuani. The CL-Atse Protocol Analyser. In Frank Pfenning, editor, *17th International Conference on Term Rewriting and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 277–286. Springer, Seattle, WA/USA, 08 12, 2006.

[72] Qurat ul Ain Nizamani and Hyder A. Nizamani. Analysis of a federated identity management protocol in soc. In *Proceedings of 3rd Young Researchers Workshop on Service Oriented Computing (YRSOC'08)*, 2008.

[73] Qurat ul Ain Nizamani and Emilio Tuosto. Heuristic Methods for Security Protocols. *Electronic Proceedings in Theoretical Computer Science*, 7, 2009.

[74] Vincent Vanackére. The TRUST protocol analyser. Automatic and Efficient Verification of Cryptographic Protocols. In *VERIFY02*, 2002.