*Technical Report*

UCAM-CL-TR-816
ISSN 1476-2986

Number 816

# Verification of security protocols based on multicast communication

Jean E. Martina

March 2012

This technical report is based on a dissertation submitted February 2011 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare College.

ISSN 1476-2986

## Abstract

Over an insecure network, agents need means to communicate securely. To these means we often call security protocols. Security protocols, although constructed over the arrangement of simple security blocks, normally target the yielding of complex goals. They seem simple at a first glance, but hide subtleties that allow them to be exploited.

One way of trying to systematically capture such subtleties is through the usage of formal methods. The maturity of some methods for protocol verification is a fact today. But these methods are still not able to capture the whole set of security protocols being designed. With the convergence to an on-line world, new security goals are proposed and new protocols need to be designed. The evolution of formal verification methods becomes a necessity to keep the pace with this ongoing development.

This thesis covers the Inductive Method and its extensions. The Inductive Method is a formalism to specify and verify security protocols based on structural induction and higher-order logic proofs. The account of our extensions comes to enable the Inductive Method to reason about non-Unicast communication and threshold cryptography.

We developed a new set of theories capable of representing the entire set of known message casting frameworks. Our theories enable the Inductive Method to reason about a whole new set of protocols. We also specified a basic abstraction of threshold cryptography as a way of proving the extensibility of the method regarding new cryptographic primitives. We showed the feasibility of our specifications by revisiting a classic protocol, now verified under our framework. Secrecy verification under a mixed environment of Multicast and Unicast was also done for a Byzantine security protocol.

# Contents

—*The beginning of wisdom, is the definition of terms.*

(Socrates)

# 1 Introduction

Cryptography as such is not enough to guarantee security in communications. The way we use cryptography is as important as the strength of the cryptographic algorithms. With this assertion, we start a thesis that is meant to talk about how to verify that we have correctly combined cryptographic primitives to achieve security goals in communications. We will talk about security communication protocols or simply security protocols applied to multicasting communication.

A security protocol is a set of rules we establish to communicate achieving security goals. Security goals are normally stated as integrity, confidentiality and authentication, but they are not limited to that. To better explain the importance of security protocols we can draw parallels with other day-by-day security issues we face. To do so, we will borrow the clever example from Casimier Cremers [48] where he traces the parallels between a real world protocol to secure a bicycle and an electronic communications protocol to secure digital assets.

Figure 1.1: A very secure Bike sculpture [a]

[a]Photo from Dustin Sacks under Creative Commons Licensing

We can compare a security protocol for protecting the access to our bank accounts over the internet to a lock and chain we use to secure our bike from being stolen. We know that to achieve a good security level, we need good components in our security scheme. But we also know that we can buy the best lock available in town, and use it with the hardest chain we can find, but it still may be insufficient. The security of our bikes will be defined not just by the qualities of our tools, but in the way we use them.

For example, if we attach our bike to something loose, even the best lock and the hardest chain are prone to fail. Our bikes can get stolen because we do not understand thetools' security requirements and we do not verify that they are available, or that they were fulfilled before claiming it was secure. Another example is attaching our bike by the front wheel. Although we verified the requirement of not attaching the bike to something loose, we misunderstood the threat model. The thief can steal it by detaching the frame from the front wheel. In this case we considered the threat as having the bike stolen as a whole, and did not take into consideration it could be disassembled and partially taken.

Another issue that affects how security requirements are met is how difficult it is to the user to implement the security device, or how difficult it can be to assure the requirements were met. Figure 1.1 shows us a secure bike, but we clearly see that an extremely secure setup implies into usability drawbacks. Secure communication protocols can suffer from similar downsides.

Having good cryptographic primitives is definitely necessary to achieve a security goal. But using them properly is as important. Understanding their requirements and the threat model they are subject to is paramount. Security protocols suffer from similar flaws to the ones explained in our bike example.

We can have a very strong protocol to guarantee confidentiality, integrity and authentication when accessing websites over the Internet. But, if we fail to verify the certificate that introduces the peer identity we are connecting to, something bad can happen. If we do not verify the existence of a pre-requisite for the protocol's security prior to its run, we may not be able to achieve its security goals. In fact, we can be assuring confidentiality, integrity and authentication with the wrong peer, giving him credentials to access our data which could be our bank account. In this scenario, the attack can be called a phishing scam.

We also have cases of security protocols that were designed to one threat model and in reality were subject to another. The bike example becomes illustrative. Any security protocol's properties should be verified against a reasonable threat model. The threat model we design and verify our protocols against should reflect the threat model from the environment which the protocol will execute from. Failure to do so can leave us with ill-designed security protocols that have inherent failures. We cite the classical case of the Needham-Schroeder Public Key distribution protocol [93, 77] as an example. The protocol is secure under the threat model that no internal peer acts maliciously. When we change the rules over which the protocol should execute, we inherently change the achievability of its security goals. This is the problem found by Lowe in his classical paper on model checking security protocols [77].

The above examples show that guaranteeing the security of communication protocols is not an easy task. The main challenges for providing such security guaranties are: the number of variables involved, the possibly infinite combinations of situations in any execution, and to understand how the threat model influences the achievement of the security guaranties.

To properly discuss security protocols we need first to get their definition right. Security protocols for computer networks are sequences of steps peers must take to achieve a security goal. In each step we are exchanging messages between the involved peers. These messages may contain identifications, shared secrets, cryptographic keys, session identification, freshness components, the concatenation of all the above and more. Message exchange is helped by the usage of cryptographic primitives to establish the existence of desired security properties. Everything in the context of security protocols must be done keeping in mind the threat model the protocol is subject to. Security protocols are distributed computations capable of being run in parallel by an unlimited number of peers, including attackers and malicious internals.

Threat model is necessary for the verification of any security claim by any protocol. Without a reasonable threat model, any protocol can be claimed correct. A widely accepted threat model for security protocol analysis is the one proposed by Dolev and Yao [51]. In this threat model we have a powerful attacker that controls the communication traffic selectively. He is able to overhear, intercept, and synthesise any message. He is only limited by the constraints of the cryptographic methods used. This attacker model seems reasonable for most applications of security protocols over the Internet.

Some research effort has focus on giving the attacker a more realistic shape. Some extensions to the Dolev-Yao model concern the attackers' cryptanalysis powers [1, 14]. It is claimed that doing so would enable the analysis to capture other security protocol subtleties, especially in the choice of cryptographic primitives. We also see research about extending the attacker model without opening the black box of cryptography. Among them we have the Rational Attacker [10] where different attackers collude on the basis of cost/benefit decisions whether to follow or not to follow the protocol. The General Attacker [10], drops the cost/benefit decision from the Rational Attacker. The General Attacker's differentiation from a Dolev-Yao is that each peer acts for his own sake. And the Multi-Attacker [11], where each principal behaves as a *Dolev-Yao* attacker, but they will never reveal their long-term secrets to other peers.

Protocol design and verification has been a well researched topic for a long time now. The protocol design area is very active in designing novel and complex protocols to achieve new security goals imposed by the information revolution we live in. The protocol verification area has also been active, since protocol designers often fail to enforce the goal they claim to achieve. For the sake of exemplifying such failures, we can cite the recent discoveries on widely deployed and extensively verified protocols, such as the re-negotiation flaws discovered in SSL/TLS [39] or also cite classical cases, such as, Lowe's attack [77] on Needham-Schroeder protocol.

During the initial phase of protocol verification in the 1980's and mid-1990's, protocol verification was carried out informally. Informal verification was important because it taught us the importance of understanding the semantics behind the protocols' messages. Due to its informality it is often easier to find and understand minor flaws by using such techniques. No complicated or extensive reasoning is usually involved, what makes the outcome easy to understand. This is still a usual way of starting the evaluation process for security protocols.

From mid-1990's, we started seeing an increased interest on the usage of formal tools to help the verification of security protocol models [86]. We can cite efforts such as: Burrows et al. Belief Logic [38], which first represented formally the beliefs that peers running the protocol can derive during the execution. The Bellare and Rogaway provable security study [27], where the security requirements are met provided the assumptions about the adversary's access to the system are satisfied, and some clearly stated assumptions about the hardness of certain computational tasks hold. Abadi and Gordon's spi-calculus [2] which is a security tailored version of the popular $\pi$-calculus [90], where processes communicate through different security channels. Ryan and Schneider's state enumeration [111], which apply the well known process calculus CSP [68] to security protocol verification.

This thesis will focus on the usage of the Inductive Method [101, 20], a formalism that allows us to prove the existence of security properties via structural induction over an unbounded protocol model. The Inductive Method is assisted by a very powerful tool called Isabelle/HOL [97] which is an interactive theorem prover for Higher Order Logic. One of the main advantages of Isabelle/HOL is its generality. This generality enables us to easily extend the modelling infrastructure to verify novel protocols, something normally constrained in specialised tools [79, 87, 8, 91, 116, 107]. The usage of Higher Order Logic allows to quantify over predicates and functions and allows us to type our functions helping us to picture protocols in a realistic and strict way.

Protocol verification techniques have been aimed to the verification of security protocols based on Unicast. The scenarios depicted by most tools are based on an idea of one-to-one peer communication. Under this model, the establishment of security goals is based on the controlled distribution of knowledge between a pair of peers under the threat of an attacker. This knowledge distribution is normally aided by standard cryptographic primitives, such as shared key and public key cryptography. Verification under a Unicast framework assures that only the two peers acquire the knowledge carried by one message in the presence of a powerful attacker.

But from the protocol designers' point of view, under the necessity of achieving novel security goals imposed by the natural evolution of our networked environment, Unicast and standard cryptographic operations are not enough. Nowadays the establishment of multi-party security goals is a necessity, and this certainly changes the way we design protocols. We have seen in the last decades the creation of a new class of protocols based on unicast, multicast, broadcast and a mixture of the three modes being developed. We can cite numerous examples, such as protocols to assure secrecy on one-to-many communications [65], protocols to guarantee authenticity in one-to-many communications [60, 125], key distribution in one-to-many communications [66, 32], and protocols that deal with novel security goals such as Byzantine agreement [47, 69, 12, 127], multi-party computation [33] and digital-rights management [37, 103].

The challenge of verifying such novel protocols based on one-to-many communications is at least threefold. By changing the communication model from a one-to-one communication to a one-to-many communications, we inherently change the way we understand protocol execution. The linearity of protocol execution traces is lost and we need a reinterpretation, since the sending of a single message now changes the knowledge-set of many peers. We also have the challenge of developing models for novel cryptographic primitives. The usage of one-to-the-many communication model asks for support of primitives that deal with thresholds [40]. For example, a necessity for achieving goals such as secrecy under Byzantine agreement scenarios, requires the existence of security primitives based on threshold cryptography. And third, we need to reinterpret the output of the verification process. Under the one-to-many communication scenario, our security requirements tend to demand more fine grained evaluation. For example, the existence of an attacker model similar to the Multi-Attacker [11] becomes much more appealing for non-unicast communications.

On the verification of novel protocols, our effort in this thesis is to verify security protocols based on multicast communication. More specifically, we would like to deal with protocols that implement a Byzantine agreement model of security [58]. The Byzantine agreement model [76] allows arbitrary behaviour of a certain fraction of peers in our distributed protocol models while still achieving the protocol's security goals. This is one of the preconditions for a secure multi-party computation [61], where a set of processes want to compute a deterministic function of all their inputs without trusting each other, and in this sense it exhibits Byzantine behaviour. The verification of Byzantine Security Protocols involves the creation of support infrastructure, such as a one-to-many communication model and the modelling of threshold security primitives.

Our aims in this thesis are to enable the Inductive Method to verify protocols based on one-to-many communication by implementing a new message model, extending its support for new cryptographic primitives and do an initial verification of secrecy for a Byzantine agreement protocol based on multicast communication.

The structure of this chapter includes the motivation of our work (§1.1) followed by a statement of our contribution to knowledge (§1.2). We then follow introducing the notations used (§1.3) and outlining the remaining chapters (§1.4).

## 1.1 Motivation

The main motivation for this thesis is to further develop the Inductive Method introduced by Paulson [101] in the late 1990's. The main idea behind the Inductive Method is the usage of structural induction to reason about properties of security protocols. A protocol model is a set of traces generated by an inductive description of protocol events and messages, for which, we prove the existence of the desired security properties. By the definition of induction, the verification process is carried out over arbitrary many executions and agents, all in the presence of a standardised (but powerful) attacker model. Proofs are based on how acquisition of knowledge by the peers is performed and on the property being verified.

During the study of the Inductive Method to verify novel security protocols, the lack of tools and methods that could encompass the use of multicast primitives became evident. Although multicast was always advertised as a scheme for better network resources usage [105], we see interest in it from the security community. Initially with protocols for secure content delivery [37, 103], and later on in protocols that involve Byzantine security [76].

Entering the area of Byzantine Security Protocols, we see the necessity of protocol verification tools and methods that support advanced security primitives. Most methods and tools [86, 22] are constrained to the usual public and shared key cryptography, lacking support for new primitives that can deal with the problem of peers being partially corruptible. As examples of such primitives, we can cite verifiable secret sharing [45, 54], signature sharing schemes [57, 41] and most threshold-based mechanisms [70].

Below we will try to better cover the motivation details. Sub-section 1.1.1 will give a better overview of the current state of the Inductive Method. Sub-section 1.1.2 will look over the motivation for better multicast communication support in protocol verification methods. Finally sub-section 1.1.3 discusses the increasing interest in Byzantine security and how the support for the needed primitives, event model and message model is important for protocol verification's future.

### 1.1.1 Develop the Expressiveness of the Inductive Method

As already stated, the Inductive Method developed by Paulson is a very powerful and extendable tool [101]. During the 1990's Paulson worked on developing the basis for the method's mechanics, which included the definition of the operational semantics behind the tool and the basic lemmas targeting the verification of classical protocols [100].

Paulson initially created a set of general theories for agents, messages, events and an extended formalisation of the attacker, here known as the *Spy*. He also proved the basic lemmas needed for the method to be usable. He used the method to prove a few classic security protocols which showed the method's potential [101]. Clearly the unbounded agent population and its ability to interleave an arbitrary number of protocol sessions were crucial features.

Later Bella worked on extending the Inductive Method by expanding the size of the protocols it was capable of verifying [18]. He worked on the basis that Paulson's work needed a bigger challenge. He successfully verified using the method a great range of real security protocols [23, 24, 19, 26, 25], such as the Kerberos family of protocols, smart-card protocols, electronic commerce protocols, and protocols that deal with accountability.

In his work [20], Bella introduced a new range of extensions to the Inductive Method, which showed its extension capabilities. His main contributions to the method are the discrete modelling of timestamps, the extension of the agent model for a smart-card scenario, the verification of novel goals such as accountability and the description of Goal Availability. Goal availability prescribes that the protocol guarantees should be based on assumptions that the protocol participants can verify by themselves during protocol execution [18, 20]. Goal availability is a key principle underlying the Inductive Method.

The motivation of this thesis is to propose extensions to the Inductive Method on event modelling and cryptographic primitive definition.

Under the area of event modelling, we have seen an increasing number of tools and methods working under the same assumption of a unicast only world [86, 22], despite increasing interest in the use of multicast and broadcast primitives in security protocol designs. Another motivation is that *unicast and broadcast are extremes for multicast communication*. They are extremes in the sense that a unicast communication can be represented by a multicast with a multicast group of size one. In parallel, broadcast communication can be represented by a multicast with a multicast group comprising of all peers present in the network. A generic model is possible and feasible. These changes in the event model can bring a new class of protocols to be verified by the Inductive Method.

Our motivation is to create a generic event model capable of representing multicast and broadcast as well as unicast. We propose a new theory representing events based in a multicast scenario and use it to verify protocols under the unicast framework for backward compatibility. We also plan to show its usability in a multicast scenario.

Under the area of cryptographic primitives, we see other methods and tools being easily adaptable and extensible to such new definitions [31, 48], but no effort was ever made with the Inductive Method to prove its extensibility in this area. The novelty introduced by the usage of multicast communication in security protocols also introduces the need for modelling new cryptographic primitives [70]. To make the Inductive Method more accessible and a real world working tool, it needs the ability to implement novel cryptographic primitives.

Our motivation is to construct in the Inductive Method support for threshold cryptography, specifically secret sharing and verifiable signature sharing. It should be done in a less intrusive way, so that it can be embedded into the protocols we are designing when needed.

By having support for a new event model and new cryptographic primitives, we can verify a larger set of security protocols. This push in the Inductive Method can help the security verification community to address the problem by also extending other tools and methods.

### 1.1.2 Lack of Proper Tools for Multicast Protocol Verification

Security protocol design until the beginning of the last decade was only focused on the development of protocols based on a one-to-one communication. Since the execution of experiments with the MBONE Network [75] and the increasing commercial availability of multicast, security protocol's designers started exploring new techniques based on one-to-many communication methods to achieve novel security goals. Due to the intrinsic complexity introduced by one-to-many communication systems, protocol designers had to experiment with new cryptographic tools to solve these new problems.

We saw the creation of a series of new protocols to achieve goals already explored under one-to-one communication. As examples we can cite protocols to assure secrecy [65], to guarantee authenticity [60, 125], and to distribute keys [66, 32]. These revisits to already well understood security goals were done to accommodate the new environment imposed by one-to-many communications. Since the guarantees are not given on a pair based situation, new considerations have been raised and addressed. We also saw the creation of new security protocols that yielded novel security goals, such as Byzantine agreement [47, 69, 12, 127], multi-party computation [33] and digital-rights management [37, 103].

Protocol verification on the other hand, evolved in a much slower and unfocused way. We saw efforts being made with different tools and methods to verify multicast security protocols, but they always hit some constraint which would be easily addressable by the Inductive Method. As examples we can cite the attempt done with the NRL Protocol Analyser [87], which suffers from state explosion problems and a poor cryptographic extensibility issue [88]. Other problems using a calculus approach can be seen on Anastasi et. al. [5]. Attempts of using SAT engines such as the Alloy Analyser [74] also hit issues regarding state explosion [122].

On the other hand, we saw a great deal of work being done on a method directly derived from the Inductive Method to extend it towards multicast protocol verification [118]. We can cite the work from Steel and Bundy [119, 118] using the automated inductive counterexample finder CORAL [120]. CORAL is a first order version of the Paulson/-Bella Inductive Method used to search counterexamples to the security properties under consideration. As a search tool, CORAL does not assert the existence of properties in protocols, but simply tries to find common errors in protocol design.

CORAL being a lightweight version of the Inductive Method uses first order logics and lacks a strict typing system. Even though it is able to bring new light to the verification of multicast based security protocols. We believe the extension of the Inductive Method towards a one-to-many communication model is capable of capturing new insights into protocol verification, now under the scenario of multicast and broadcast communication.

Our motivation, as previously stated, is the creation of a robust events theory for the Inductive Method that can encompass the verification of unicast, multicast and broadcast security protocols under the same infrastructure. We should be able to verify protocols that use one-to-many communications and re-interpret their results under the existing framework of the Inductive Method. This ultimately can help us to advance understanding of the suitability of new attacker models and also the formalisation of novel security goals.

### 1.1.3 Byzantine Security

With the evolution of the Internet and the creation of an environment based on a scheme where no one can be one hundred per cent trusted, we see the increased interest for protocols that are able to cope with partial corruption of peers and still achieve its goals. Such protocols are based on the idea of Byzantine Agreement Security [59].

In the original Byzantine Generals Problem [76] the participants are generals of the Byzantine army who want to agree of whether to attack an enemy town or not. The generals are either loyal or traitors. Traitors can act in arbitrary ways, and, for example, they can send conflicting messages or fail to participate in the decision process. This model is very handy to help us model the actual framework over which the Internet works: We cannot trust our peers and we cannot trust the network.

Byzantine agreement security introduces new challenges to the protocol design and verification process. Novel goals can be achieved by using Byzantine Security Protocols, such as, reliability of the information dispersed among peers and the availability of the information under corrupted schemes. Reliability is normally achieved by sending the information to multiple peers by using multicasting techniques. But, the original Byzantine Generals problem does not account for secrecy since its main goals are to achieve reliability and availability. This is one of the main concerns when designing security protocols based on Byzantine agreement [12, 59].

Secrecy of one peer's information is not useful for an attacker, since the protocol is built to cope with this corruption up to a threshold. On the other hand, secrecy of group based information must be handled by non-standard cryptographic primitives based on thresholds, such as secret-sharing [70] and verifiable signature sharing [57]. The usage of such cryptographic primitives brings new challenges to protocol verification.

Our motivation regarding Byzantine Security Protocols is to be able to verify protocols with an arbitrary number of corrupted peers and assert the existence of basic security properties, such as integrity and secrecy, using new cryptographic primitives. We are not focusing now on the establishment of the novel verification goals, but on the impact of changing the underlying casting primitive has on achieveing reliability and secrecy.

We have chosen the Franklin-Reiter Secure Auction Service [58] as test bed for our proposed innovations, because of its composition as a true Byzantine agreement security protocol. It uses multicast and unicast for achieving reliability, secret sharing and verifiable signature sharing for achieving integrity and secrecy, as well as it introduces novel security goals for future experimentation, such as partial-secrecy and a weak form of anonymity.

## 1.2 Contribution

Our contribution can be classified in three main groups: The extension of the Inductive Method to enable the usage of a one-to-many event model (§1.2.1), the specification of a base formalism to encapsulate new cryptographic primitives (§1.2.2) and the verification of partial secrecy under a mixed event model Byzantine Security scenario (§1.2.3).

### 1.2.1 Event model for Multicast, Broadcast and Unicast

As observed by Bella [20], the Inductive Method turns out to be easily extendible. The extension of the event model takes into account that communication was not done anymore on a one-to-one fashion. This new direction brought us challenges in terms of the already available infrastructure in the Inductive Method.

We were able to implement a usable multicast theory that is capable of implementing protocols based on the three known methods of communication. This extension enables us to reason about different classes of multicast. With it we were able to represent Atomic and Reliable multicasts, as defined in Section 3.1.

We also did one experiment to corroborate our objective of creating a new underlying event framework that was capable of representing all the communication casting methods available today. We revisited Needham-Schroeder Shared Key Protocol re-specified under the new event framework. The result of this experiment corroborated that it is not unreasonable, for the sake of broader coverage, to impose to the Inductive Method the usage of the new events framework as a standard. It does not further complicate the method, but simply requires some intermediate commands under Isabelle/HOL to achieve the goals.

Another achievement of our research is the modelling of protocols that operate under different message casting frameworks in different phases. We did that with the Franklin-Reiter Sealed Bid Auction Protocol and we were able to integrate both casting frameworks under the same knowledge distribution model. This brings the Inductive Method to a new scope of coverage in terms of what protocols can be specified and verified by it.

### 1.2.2 Implementation of a base formalism for secret sharing

The addition of new cryptographic primitives under the Inductive Method is not straightforward and generally breaks the compatibility with past verifications done with it. Our main contribution was the extension of the Inductive Method to accept primitives based on threshold cryptography by shielding them with nonce proprieties and developing a basic abstract model of their operation. In this sense we were able to extend the method by adding such new cryptographic primitives by just dividing the space occupied by nonce and taking into consideration that the secret sharing mechanisms we opted to develop are computationally secure, as is all cryptography in the Inductive Method.

The novelty introduced regards also the experimentation within the Inductive Method with non standard cryptographic primitives and its application under a protocol verification scenario. Some other efforts tried to formalise and propose some verification for stand-alone secret sharing [126].

We must stress that our study is not thorough nor complete, but a starting point for the representation of such new class of security primitives. The representation of the mechanics behind access structures present in the variety of threshold schemes was not achieved, but is clearly the next step to be done in extending the Inductive Method towards new security primitives.

### 1.2.3 Initial Verification of the FR-Auction Protocol

As our main target in this thesis was the specification of a capable and workable multicast theory, doing the initial verification of security protocols that demanded the usage of such a specification was important. We opted for doing an initial secrecy verification of threshold cryptography components in a protocol that made use of Byzantine agreement and a dual message casting framework.

As our contribution under this topic, we can cite the specification and verification of partial confidentiality under a mixed unicast-multicast environment that was achieved for the Franklin-Reiter sealed bid auction protocol. We did a full specification of the Franklin-Reiter sealed bid auction protocol under our new proposal of multicast events for the Inductive Method, and made use of our initial specification of threshold cryptography mechanisms. In the verification side, we proved some reliability lemmas, and the partial secrecy required for the shared components. We also introduced the required formalisation to deal with unbounded bid-sets present in each session of the protocol.

We are not claiming to achieve a complete and thorough verification of the Franklin-Reiter sealed bid auction protocol, but establishing the basic properties needed for the verification of secrecy in components being shared under the threshold cryptography proposed specification and under a multicast environment.

## 1.3 Notation

To be able to properly discuss properties on security protocols, we must establish the basis for the notation representing them in this thesis. The notations we chose to adhere started with the early paper from Needham and Schroeder [93] and were refined over time. For the sake of easier readability for people interested in the Inductive Method, the notation will be kept similar to that used by Paulson [101] and Bella [20] in previous works, extending it where needed.

### 1.3.1 Protocol Notation

The traditional protocol description method cited above consists of describing each step of the protocols in one line. Each line is numbered and shows the direction of the information flow, which is generally from left to right. The sender is named on the left of the operation and the receiver is named on the right.

Each step is composed of three fields: the numbering separated by a "dot" from the message type/flow direction, which is separated by a semicolon ":" from the message payload. Components in the message payload are separated by a comma ", and grouped using fat braces "{|" and "|}". Fat Braces are generally used by the operation of a function of a cryptographic primitive over the grouped message components

Protocols throughout this thesis will be presented in a similar way as the example protocol shown in Figure 1.2.

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 1. | $A$ | $\longrightarrow$ | $B$ | : | $A, Na$ |
| 2. | $A$ | $\overset{B}{\rightsquigarrow}$ | $BG$ | : | $\lambda X.\{\lvert A, Na \rvert\}_{eK_X}$ |
| 3. | $B$ | $\overset{R}{\rightsquigarrow}$ | $M$ | : | $\lambda X.\{\lvert Na, \{\lvert Nb \rvert\}_{sK_B^-} \rvert\}_{eK_X}$ |

Figure 1.2: Example notation protocol

This protocol consists of three steps in a mixed unicast, broadcast and multicast environment. In the first step, using the standard known notation, agent $A$ sends to agent $B$ using the flat arrow ”$\longrightarrow$”, a two component message composed by his identity $A$ concatenated with a special number *Na*. *Na* is a nonce generated by $A$. By definition a nonce is a random number that is used only once. In the context of security protocols it may confirm the freshness of a message in relation to when the nonce was generated. It also plays an important role in session control and authentication in the protocol.

The second step shows a broadcast message from agent $A$ to the broadcast Group *BG*. The message is the compact representation of the actual transmitted data. The broadcast message is represented by the curly arrow with the letter *”B”* on the top, E.g. ”$\overset{B}{\rightsquigarrow}$”. In this message we see the application of the function $\lambda X.\{\lvert A, Na \rvert\}_{eK_X}$ to all the components in the payload and where $X$ represents each Agent member of the group. The application of this function represents the individual view of each Agent $X$ belonging to the Broadcast Group *BG*. We could read this step as: Agent *”A”* sends a broadcast message to the broadcast group *BG* containing his own identity and a nonce *Na*, individually encrypted to each broadcast group member's public encryption key $eK_X$.

On the third step we see Reliable multicast from the Agent $B$ to the multicast group *M*. Multicast messages are represented by the use of a curly arrow with its type written on the top of the arrow. For example, an Unreliable multicast is shown as ”$\overset{U}{\rightsquigarrow}$”, a Reliable multicast as ”$\overset{R}{\rightsquigarrow}$ ”, and an Atomic multicast as ”$\overset{A}{\rightsquigarrow}$ ”. The payload field follows the same structure as in the broadcast message, which makes the message in step three read as follows: Agent $B$ sends a Reliable multicast to the multicast group *M*, where by the application of the function $\lambda X.\{\lvert Na, \{\lvert Nb \rvert\}_{sK_B^-} \rvert\}_{eK_X}$ to all the components in the payload and where $X$ represents each Agent member of the group, the nonce *Na* concatenated with the nonce *Nb* signed with the private signing key of $B$ is sent individually encrypted to each member of the multicast group using each member's public encryption key $eK_X$.

As noted by Bella [20], this notation is only fully understandable when put in the context of the threat model it is being used with. In the threat model generally used by protocol designers [51], the *Spy* can intercept all messages and prevent their delivery. He can tamper with the messages by looking over everything he has enough knowledge to open. The *Spy* can also generate new messages from what he knows or has. The attacker cannot break cryptographic algorithms by the use of brute force or cryptanalysis. Bella's work [20] also focused on the idea of message reception and the derivable knowledge in it, which is not represented by this notation.

### 1.3.2 Logical Notation

As this work consists of applied logical expressions, it is important to describe the English equivalent for each logical symbol being used throughout the thesis. The idea is to help the reader into grasping the logical symbols when they are not written in plain English. They should be read as:

| Logical Context | Symbol | Reads in English |
|---|---|---|
| conjunction | $\wedge$ | "and" |
| disjunction | $\vee$ | "or" |
| negation | $\neg$ | "not" |
| equality | $=$ | "is equal to" |
| disequality | $\neq$ | "is not" |
| equivalence | $\Longleftrightarrow$ | "in and only if" |
| meta-level implication | $[\vert \ldots \vert] \Longrightarrow \ldots$ | "if ... then ..." |
| existential quantification | $\exists$ | "for some" |
| universal quantification | $\forall$ | "for any" |
| set membership | $\in$ | "is in" |
| negation of set membership | $\notin$ | "is not in" |
| subset condition | $\subset$ | "is a subset of" |
| set-theoretic union | $\cup$ | "the union of" |

Table 1.1: Logical Notation

English and the logical symbols will be interchangeable in the text, the English preferably used for the sake of comprehension and the symbols for the sake of compact representation. As the Isabelle theorem prover offers support for the mathematical symbols internally through the use of UTF-8 characters [97], most theorems output from it will use the symbolic notation.

## 1.4 Thesis Outline

We briefly sketch the organisation of the thesis, and summarise the contributions made in each chapter.

### 1.4.1 Chapter 2: Unicast Protocol Verification

In Chapter 2 we present the current state of protocol verification techniques. We list the available methods, dividing them into informal and formal approaches, giving brief descriptions of their evolution and relation. We also classify these methods by construction, trying to give some interpretation on their advantages and disadvantages for the formal specification and verification process. Finally we close the chapter with a deep presentation of the Inductive Method. The main contributions are a fresh literature review of the area of protocol verification and an up-to-date presentation of the Inductive Method.

### 1.4.2 Chapter 3: Security Multicast Protocol Verification

Chapter 3 starts presenting multicast Communications and its definitions. We also survey the area in terms of multicast security protocols and the new interpretation of security requirements for them. We then present our specification in Isabelle/HOL of a multicast event model that is capable of representing unicast and broadcast communications. We adhere to the idea that "unicast and broadcast are extremes for multicast communication". The main lemmas to support this model are also discussed. We conclude the chapter with a revisited verification of Needham-Schroeder's Shared Key Protocol [93], now implemented using the new multicast event model. The main contributions in this chapter are the literature review and survey regarding the multicast security protocols and the Inductive Method's extension to support such protocols.

### 1.4.3 Chapter 4: Secret Sharing Formalisation

In chapter 4 we start presenting Threshold Cryptography mechanisms, focusing especially on secret-sharing and verifiable signature sharing. We introduce the reader to a literature review of these primitives and draw our approach in their basic functionality specification. We then discuss how to extend the Inductive Method's message model without breaking backwards compatibility. The formalisation of such novel primitives is necessary for the initial verification of the Franklin-Reiter Auction Protocol [58] examined in Chapter 5. The main contributions of this chapter are the study of novel cryptographic primitives needed by Byzantine security systems and its impact on the Inductive Method's message model, together with a simple and initial specification of such primitives in Isabelle/HOL.

### 1.4.4 Chapter 5: Verifying a Byzantine Security Protocol Based on Multicast

Chapter 5 has as its core the verification of the Franklin-Reiter Auction Protocol [58]. We start discussing the importance of Byzantine Security Protocols and their close relation to multicast specification and novel cryptographic primitives. We then show the protocol specification and its semantics, focusing in the new multicast model proposed in Chapter 3. We then present some initial validation proofs, focusing on the new challenges imposed by the usage of a multicast model in terms of concluding the proofs. We then discuss the reinterpretation of findings tracing parallels with the unicast world, trying to show the increased representation brought by the multicast model.

### 1.4.5 Chapter 6: Final Remarks

Finally Chapter 6 discusses the goals achieved in the thesis, giving emphasis on the reinterpretation of the idea of traces and how security guaranties can be assessed in the new environment. We also discuss potential outcomes in protocol verification of the modified threat model imposed by any multicast security protocol. We discuss some future outcomes from our work, such as the inherent need for new threat models and the addressing of novel security goals imposed by the new threat models and the variety of message casting models.

—*Two monologues do not make a dialogue.*

Jeff Daly

# 2 Unicast Protocol Verification

During the last thirty years we have seen a lot of effort being put into designing security protocols. Due to their intrinsic complexity it is usual to have problems with the claims such protocols make in terms of security properties they achieve. This happens since it is not always clear or straightforward to understand or believe that a protocol achieves such goals. We start this chapter revising the literature and the method we will use throughout this thesis stating that analysing such claims demands a thorough approach to the problem. We can also state that the design of security protocols is error-prone, especially because it is very difficult to anticipate what is achievable by an attacker interacting through an unbounded number of protocol runs, collecting information and sometimes even acting honestly. Keeping this inherent complexity in mind, this chapter has the aim of doing a survey on the state of protocol design and protocol verification today. We named this chapter as Unicast protocol verification because our idea is to summarise all efforts done so far by revisiting the basis of our research. We also would like to clearly show the novelty of our research.

To properly understand the basis of our research we will be first establishing the principles for security protocols and how they are represented and designed (§2.1). Then we will be taking a look into protocol attacks (§2.2), which are the non adherence of the protocol to its claimed security properties. To be able to understand attacks, we need to characterise what are protocol goals and what they mean in the context of security protocol design and verification (§2.2.1). Protocol goals are only applicable if we define an attacker and a threat model, which characterise the environmental threats protocols are subject to (§2.2.2). We will see that the adherence to a threat model is paramount to the verification of any claim, since the threat model is what shapes the vulnerabilities to an attack. Understanding the attacker and the threat model we will look over classical attacks to protocols (§2.2.3) which will give us the basis to enter the discussion why we should formalise and verify them.

At the formal verification section (§2.3) we will be first discussing how the thorough review brought by formal methods can help the verification process of security protocols and why it is not only a recommended, but also a mandatory step in protocol design. To help the understanding of the effort made so far in terms of formalisation and verification of security protocols we will be looking to the approaches developed over time, trying to address the problem imposed by the triad of limitations we have in the verification process for security protocols. The triad can be defined as the limitation in the number of agents, the limitation in number of parallel sessions and the limitation in the amount of data principals can acquire during infinite runs. In this section we will be looking over the initial ideas of formalisation of security protocols (§2.3.1) and into the two main branches of protocol verification, which are State Exploration (§2.3.2) with its promising Model Checking (§2.3.2.2) and Strand Spaces (§2.3.2.3), and onto Theorem Proving (§2.3.3), showing the advantages and disadvantages of using either Higher-Order Logic (§2.3.3.1), and First-Order Logic (§2.3.3.1).

Finally this chapter will also cover a detailed review of Paulson's Inductive Method (§2.4), which was chosen to be the target for the work proposed in this thesis for not suffering the triad of limitations stated above. We will explore Paulson's Inductive Method construction by looking over the theories and definitions that make up the method (§2.4.1). The method is composed by different development branches. We will cover all the specification and assembly of the method within its main branch. With the method specified we will be looking over Bella's Goal availability (§2.4.2.1) principle since it is key for the statement of meaningful verification lemmas, as well as it can help us to uncover more subtleties of protocol properties by a thorough approach. Then we will be looking over the verification process in the Inductive Method (2.4.2). We will try to cover all the protocol goals discussed before giving the reader examples of how to state formally these properties, including or not the principle of Goal Availability.

We will conclude the chapter with a discussion of how the verification of security protocols stands today, trying to give our view and bring some extra motivation for the extension of the inductive method. We will try also to prepare the reader for our contribution, making a summary of what was achieved before our contribution.

## 2.1 Building Blocks for Security Protocols

The best way of starting to describe the problem of designing security protocols is giving context to small operational problems we have. Let's suppose *Alice* wants to send her friend *Bob* an encrypted message using an insecure network. *Alice* and *Bob* have access to cryptographic functions. These functions are key-operated and considered perfectly secure form their point of view. This is the general setup for the design of a security protocol.

Contextualising, a cryptographic function operates by taking a plain text $P$ and a key $K$ and transforming these parameters into a ciphered text $C$ by the usage of transformations and permutations. The ciphered text $C$ is unintelligible to anyone overseeing the medium, and the process of applying this function in this direction is called encryption. Upon receiving the ciphertext $C$, Bob can reverse the cryptographic function by feeding it with the ciphertext $C$ and the inverse of key $K$, normally written $K_{-1}$. The result of the application of the cryptographic function in this direction is the recovery of the plaintext $P$ back as Alice meant it to be. This last process of using the cryptographic function is called decryption.

Cryptographic functions can be classified according to the way they do their encryption and decryption regarding the usage of keys. If keys in both sides of the encryption/decryption process are different we call the function *asymmetric cryptographic function.* A *symmetric cryptographic function* is a function where $K = K_{-1}$. It is the same in both sides of the encryption-decryption process. The availability of such cryptographic functions is vast, and the assertion of security properties for them is a science in itself. We can cite examples such as DES [92], AES [55], RSA [106] and Elgamal [52].

The usage of cryptographic functions clearly requires one preparation step before their use: key provisioning. As seen before we consider a cryptographic function secure, but its security is not only determined by the neatness of the operations that happen inside the cryptographic primitive, but is directly related to how we manage the key used to configure such functions. In a symmetric scenario, a key provisioned for communication between *Alice* and *Bob* will be conventionally called $K_{AB}$. The provisioning of such key can be done in various ways. This can be by previous in-person encounter, by the delegation from a trusted third party, by an agreement or by a key establishment protocol. The asymmetric scenario requires *Alice* and *Bob* to have a key pair. In such key pair, *Alice* and *Bob* will have a key each that they will keep secure and we call this a *private key.* They will also have a key that is derived from the first one that they openly publish and we call this a *public key.*

Although a naive comparison between both types of encryption functions can state that asymmetric cryptographic functions brings us more capabilities in terms of properties we can represent with them, they suffer some drawbacks. Asymmetric cryptographic functions, also known as public key cryptography, require much larger key sizes due to their construction principles, and also are much more computationally intensive due to the sort of operations happening within the cryptographic function boundaries. Symmetric cryptographic functions, or shared key cryptography, on the other hand, use smaller key sizes to achieve the same robustness and are much less computationally intensive than public key cryptography. A general aim is to try to get the best of each breed of functions. We want fast operations and at the same time the possibility of representing the complex and important properties required by the real world complexities. The existence of two different types of encryption schemes with complementing features and drawbacks results ultimately in compositional schemes. Security communication protocols are one example of such compositional schemes, where we try to achieve a security goal using the least resources.

As discussed before, security protocols aim to allow two or more principals to establish a security goal. The properties we normally desire in such protocols include from authentication, confidentiality, integrity to non-repudiation, privacy and accountability. These properties are normally limited only by the necessity we have in the real world of them.

Cryptographic protocols were first proposed by Needham and Schroeder in their seminal 1978 paper [93]. They were trying to address the key establishment problems explained above by proposing two different protocols for interactive secure communication. One was developed using a symmetric cryptography and a trusted third party and a second one was developed based on public key cryptography and no external arbitration. They were also responsible for proposing the notation which was explained previously in Section 1.3.

To help exemplifying why secure protocol design and ultimately secure protocol verification are intrinsically complex activities we will use Needham-Schroeder Public Key protocol (NSPK) as an example. The aim of this protocol is to establish mutual authentication and key distribution between the parts executing the protocol by using public key cryptography. The protocol assumes each peer already possesses a key pair which the public component is available to all participants, including the attacker. The protocol runs like this:

$$\boxed{\begin{array}{lcccl} 1. & A & \rightarrow & B & : & \{|Na, A|\}_{pubK_B} \\ 2. & B & \rightarrow & A & : & \{|Na, Nb|\}_{pubK_A} \\ 3. & A & \rightarrow & B & : & \{|Nb|\}_{pubK_B} \end{array}}$$

Figure 2.1: Needham-Schroeder Public Key Protocol

Figure 2.1 can be informally described by the following three points, being one for each message of the protocol:

1. *Alice* wants to initiate a session with *Bob*, and to do so she generates a random number only she knows and that will be used only once. She then sends this "number usable only once" (Nonce) to *Bob* concatenated with her identity and wrapped by encryption with *Bob's* public key.

2. *Bob* then receives message one and is able to decrypt it using his private key. He learns then the nonce *Alice* sent and generates one of his own. He then sends a new message containing *Alice's* nonce concatenated with his own back to *Alice*. All message components are wrapped by encryption with *Alice's* public key.

3. *Alice* receives message two and is able to decrypt it using her private key. She then checks that the nonce in it is indeed the nonce she generated. She knows that only Bob could have returned it, because she used it only in the composition of the first message that was wrapped with encryption under *Bob's* public key. *Alice* will then use the nonce from *Bob* to create message three. Since *Bob* used that nonce only on message two, and it was encrypted under *Alice's* public key, he knows it could have come from *Alice* and that she learned that value.

After the protocol run, both *Alice* and *Bob* know the nonces *Na* and *Nb* and can use them for further encrypted communication, since they believe the other side also learnt it. NSPK was designed to establish mutual authentication and key distribution, but not just this, it also demonstrate aliveness and provides freshness. If *Alice* receives a message encrypted with the nonce *Nb* she believes not only that it came from *Bob* but that it was sent after the generation of her nonce.

Although the idea of accepting a nonce as fresh and secure based on the fact that only the intended recipient could have read the originating transmission seems sound, we will see that this protocol is flawed. In the next section (§2.2) we will talk about protocol attacks and what exactly is claimed in terms of security, specially taking into consideration protocol goals (§2.2.1). We will later see how the attacker is shaped and placed within the communication infrastructure (§2.2.2) so that we can understand the usual threat model protocols are subject to.

## 2.2 Protocol Attacks

The properties we want our protocols to have are known as protocol goals. Such goals are achieved or not depending on the threat model we use our protocols with. The relation between protocol goals and threat models is very intimate, since under a weak and unrealistic threat model, any protocols has their goals valid. This is why they are presented under the same umbrella of protocol attacks.

### 2.2.1 Protocol Goals

Protocol goals are the properties we would like our protocols to achieve during their execution. The range of goals we normally have in security protocols is broad, so we try to exemplify some here:

- *Freshness* concerns the time of creation for a message or component in a protocol. The understanding of freshness is important in protocols because it will help us to avoid attacks regarding the timing of messages. It does not depend on other security goals, since the achievement of such property is made by checking that the exact component was not used before in the protocol lifespan or that it is still valid under our conditions.

- *Confidentiality* concerns the controlled disclosure of information. In security protocols' terms we can state a component has confidentially if the Spy, or any other non intended recipient, does not learn the value of such component by any means during the protocol's run.

- *Authentication* relates the origin of a message or component with a certain agent within the protocol. Authentication is intrinsically related to Integrity, which by itself, is related to freshness. We cannot achieve Authentication of a message or component if it was tampered with or if it is out of date.

- *Key Distribution* concerns the knowledge of an agreed key by two agents after a protocol run. In other terms, key distribution is achieved when, after the run of a protocol, the involved agents learnt, and know that their peers also learnt, a key for further secure communication.

- *Non-Repudiation* concerns the impossibility of a peer to plausibly deny the participation in a protocol run and in the achievement of another goal. This property is very important especially because it directly relates to the binding of actions of a peer in the achievement of a goal.

- *Privacy* is a property that regards the deniability on the participation in a protocols run and the impossibility of relating the peer participation with its identity if he does not want so. Privacy can be seen as the counter property for Non-Repudiation.

The above list is not exhaustive. Also the exact meaning of these high-level goals is open to a certain amount of interpretation. This makes the achievement of some of these properties an exercise of interpretation and adaptation of the claim. There has been some effort in the precise statement of such properties, since this helps the understanding and the confirmation of claims security protocols have. We can cite work from Lowe [78], Goolmann [62] and Bella [20].

Understanding goals we want security protocols to achieve is paramount for verifying any claims protocol designers make regarding their creations. This understanding coupled with the understanding of threat models protocols are subject to are key to verify any claim.

### 2.2.2 Threat Model and the Spy

The threat model can be seen as the environment where we are claiming our security properties to hold in a security protocol. A threat model is generally related to the idea of a potential attacker to our protocols and his capabilities. Such a *Spy*, as we can also call the attacker, was first mentioned in the classical Needham-Schroeder paper [93]. Needham and Schroeder made assumptions about the behaviour of principals executing the protocols and the potential attacker they were aiming to protect against. Later this definition became a central point in discussing if the protocols were correct or not, and definitely demonstrated how the security claims and their verification are tied to the threat model to which they are subject.

Needham and Schroeder made assumptions that are commonly accepted by the security community, are up to date and usable even with the new shape the Internet gave to computer networks. Their assumptions are:

- Cryptography in security protocols is perfect and not breakable by the use of brute force or cryptanalysis.

- The attacker can manipulate all communication paths. He can oversee all traffic as well as, delay, prevent delivery and fake messages of his own using all resources his has available, with exception to cryptanalysis powers what would be contrary the first point.

- Principals on the network other than the attacker are following the protocols, and security protocols do not force all the communication to be carried out in a secure fashion.

Under the evolution of computer networks we see new additions to bring the threat model closer to the real threat protocols would be subject to. Dolev and Yao [51] formalised their attacker with the previous capabilities but added some further important assumptions:

- The attacker can break down messages up to their atomic components and decrypt all encrypted messages to which he possesses the key.

- The attacker can forward encrypted messages he cannot read.

- The attacker can be an internal agent to the protocol that engages in a run to learn information and use it later to leverage gain.

The threat model known as *Dolev-Yao* is today the standard in terms of security protocols research. From the *Dolev-Yao* threat model we see the spark of two different research lines. The first research line agrees with the threat model in terms of architecture and tries to extend it with probabilistic and cryptanalysis powers. Their main motivation is that by doing such extensions the threat model would depict an even more realistic threat scenario. This has been focus of interest since Bellare and Rogaway's [27] worked on computational complexity and the threat model.

The second line of interest in threat modelling description believes that subtleties of protocols attacks can still be discovered by rearranging the power distribution between the omnipresent and all-powerful *Dolev-Yao* attacker and other potentially interested subjects. This new line of research brings us more realistic scenarios from protocols to adhere in the sense threat today is different from the cold war one historically embedded in the *Dolev-Yao*.

We tend to focus more on this second line of research since it is easier to implement into the symbolic protocols analysis, which this thesis focuses. An example is the *BUG* threat model [21], where the agents participating in the protocol are divided in three groups: the Good, the Bad and the Ugly. They are agents that follow, subvert or change behaviour respectively during the protocol execution. The *BUG* threat model is important because of its novelty in having attackers not sharing their knowledge and changing their behaviour during the run. As direct derivations from the *BUG* threat model we can also cite the *Rational Attacker* [10] where the agents make cost/benefit decisions on when to behave or not, and the *General Attacker* [10] where the cost/benefit function is dropped, but principals still do not collude as in the original *Dolev-Yao*. Finally in recent research [11] we have the introduction of the *Multi-Attacker*, which is a variant of the BUG family where each principal may behave as a *Dolev-Yao* attacker but will never share his long term secrets with other agents.

The advantages of evolving the original *Dolev-Yao* attacker into forms like the *BUG* family of threat models is that these properties are more adequate to their actual execution environment. The possibility of retaliation or anticipation attacks as suggested by Arsac ET al. [11] justifies the consideration of such new threat models.

As one of the focuses of this thesis is the understanding how Multicast communication affects security protocols, the evaluation against some of these novel threat models was considered. Especially regarding the evolution of the *Dolev-Yao* towards the *Multi-Attacker*. The scenario of a *Multi-Attacker* could bring insights on how the knowledge is distributed in a Multicast setup among peers. If we assume a *Multi-Attacker* threat model then the reflection of knowledge between peers during Multicast that will be explained in Section 3.2.1 makes sense. We did not implement the *Multi-Attacker* threat model because we dropped the implementation of the reflection of knowledge, as will be later explained.

Summarising, understanding protocol goals and threat models is key to understanding attacks on security protocols. Although simple in description, a threat model hides subtleties that can validate or invalidate most claims made regarding the achievement of security goals. Attacks can be seen as the marriage between weak goal achievements and misunderstanding of the correct threat model. In the next section we will be covering some of the attacks discovered against classical protocols.

### 2.2.3 Classical Attack examples

To exemplify why the understanding of security goals and their threat models is important to the analysis of claims made by protocol designers, we show probably the most notorious attack ever. This attack was discovered 16 years after the original publication of the protocol and was only feasible because of a change in the threat model.

We will cover the attack found by Gavin Lowe [77] in 1995 on the Needham-Schroeder Public Key Protocol, which we previously described in Figure 2.1. The attack is only possible because of a change in the threat model originally conceived by Needham and Schroeder. In this new threat model we accept that a dishonest player can be considered as an honest player by other principals. In other terms, we consider a corrupted internal as part of our threat model.

| | | | | | |
|---|---|---|---|---|---|
| 1. | $A$ | $\rightarrow$ | $C$ | : | $\{\|Na, A\|\}_{pubK_C}$ |
| 1'. | $C_A$ | $\rightarrow$ | $B$ | : | $\{\|Na, A\|\}_{pubK_B}$ |
| 2'. | $B$ | $\rightarrow$ | $C_A$ | : | $\{\|Na, Nb\|\}_{pubK_A}$ |
| 2. | $C$ | $\rightarrow$ | $A$ | : | $\{\|Na, Nb\|\}_{pubK_A}$ |
| 3. | $A$ | $\rightarrow$ | $C$ | : | $\{\|Nb\|\}_{pubK_C}$ |
| 3'. | $C_A$ | $\rightarrow$ | $B$ | : | $\{\|Nb\|\}_{pubK_B}$ |

Figure 2.2: Needham-Schroeder Public Key Protocol with Lowe's Attack

A corrupted internal being a plausible assumption, we see that when *Alice* starts a run with the attacker *Charlie*, he starts a parallel run of the protocol with *Bob*. In this parallel session he masquerades as *Alice* and will use *Alice* as an oracle to decrypt *Bob's* messages. Figure 2.2 shows the interleaved sessions, which are differentiated by the prime ' symbol. As a notational syntax, when we use $C_A$ we indicate *Charlie* impersonating *Alice* in the protocol.

The attack is valid because at the end of this sequence of events *Bob* believes he carried out a complete run of the protocol with *Alice*, when in fact he has not. Now *Charlie* can use the values of the nonces *Na* and *Nb* to impersonate Alice to Bob. As Lowe suggest, the following message is valid:

$C_A \rightarrow B : \{|Na, Nb, \text{transfer thousands of £ to C }|\}_{pubK_B}$

Although not designed to be a banking protocol it shows clearly a flaw in authentication. This attack at time of discovery caused a lot of controversy, especially due to the fact Lowe changed the threat model to make it plausible. The attack was widely accepted because the threat model used clearly showed the limitations of the protocol in a real threat scenario where the attacker can in fact be an internal.

The second protocol in Needham-Schroeder's paper [93], shown in Figure 2.3 , was also subject to an attack [50]. Denning and Sacco showed the presence of a freshness problem or *replay attack*. This happens when the attacker re-sends a message an honest agent sent earlier and proceeds with the protocol.

The Needham-Schroeder Shared Key protocol starts with *Alice* sending a message to a trusted third party called *Steve*, where she indicates she wants to talk to *Bob*. She includes in this first message a nonce generated by herself plus her identity and *Bob's*

| | | | | | |
|---|---|---|---|---|---|
| 1. | $A$ | $\rightarrow$ | $S$ | : | $\{A, B, Na\}$ |
| 2. | $S$ | $\rightarrow$ | $A$ | : | $\{\vert Na, B, K_{AB}, \{\vert K_{AB}, A\vert\}_{K_B}\vert\}_{K_A}$ |
| 3. | $A$ | $\rightarrow$ | $B$ | : | $\{\vert K_{AB}, A\vert\}_{K_B}$ |
| 4. | $B$ | $\rightarrow$ | $A$ | : | $\{\vert Nb\vert\}_{K_{AB}}$ |
| 5. | $A$ | $\rightarrow$ | $B$ | : | $\{\vert Nb - 1\vert\}_{K_{AB}}$ |

Figure 2.3: Needham-Schroeder Shared Key Protocol

identity. On message two, *Steve* will reply *Alice* with an encrypted message using the key he shares with *Alice* containing her freshness value *Na*, the identity of *Bob* and a fresh session key *Alice* can use to talk with *Bob*. *Steve* also sends a certificate to *Alice* which she will forward to *Bob*. The certificate is encrypted under the key *Bob* shares with *Steve*, and is also wrapped by the first encryption layer with *Alice*.

*Alice* then unwraps the first layer of encryption, learns the fresh session key and forwards the certificate to *Bob* in message three. Upon reception, *Bob* learns the session key *Steve* generated for him and *Alice*. He then generates a nonce for himself, and encrypts it under the session key he now shares with *Alice*. He then sends message four. When *Alice* receives message four she then decrypts it with the sessions key, modifies the nonce sent by *Bob* in a pre-determined fashion and re-encrypts again with the session key. She then sends to *Bob* message 5.

After the protocol run is concluded, *Alice* and *Bob* know the session key generated by *Steve* and believe the aliveness of each other by the reply of their nonces. However, Denning and Sacco [50] were able to show that the protocol is susceptible to a replay attack. Suppose now an attacker called *Charlie* has obtained the session key $K_{AB}$. The protocol has no restriction on the lifetime of session keys, so this should be an allowed supposition in any future point in time. If this happens, we would need to setup new session keys. *Charlie* can now fool *Bob* into using that key for a new conversation span and masquerade as *Alice* by the sequence of messages shown in Figure 2.4.

| | | | | | |
|---|---|---|---|---|---|
| 3. | $C_A$ | $\rightarrow$ | $B$ | : | $\{\vert K_{AB}, A\vert\}_{K_B}$ |
| 4. | $B$ | $\rightarrow$ | $C_A$ | : | $\{\vert Nb\vert\}_{K_{AB}}$ |
| 5. | $C_A$ | $\rightarrow$ | $B$ | : | $\{\vert Nb - 1\vert\}_{K_{AB}}$ |

Figure 2.4: Needham-Schroeder Shared Key Protocol

*Charlie* does not need to know *Bob's* long term key shared with *Steve* to be able to create message three, and he just replays it as *Alice* must have sent it as a genuine message three earlier. Then at the end of this sequence, *Bob* believes a full run of the protocol run has happened again and accepts $K_{AB}$ as a valid new session key.

The fix proposed by Denning and Sacco regarded the substitution of nonces by timestamps in the Needham-Schroeder version so that sessions keys could be expired based on an agreed duration for them. Needham and Schroeder suggest their own fix by involving an extra handshake at the beginning to avoid the clock synchronisation problems [94]. Although Denning and Sacco version is the basis for the well-known *Kerberos* family of protocols, the issue of synchronised clocks remains controversial, especially since Gong [63] exhibited the risks of relying on that.

Our third and last example of classical protocol attacks is the simplified version of the shared key protocol proposed by Otway and Rees [98] which was designed deliberately to avoid replay attacks. The main difference of this protocol from Needham-Schroeder's Shared Key one is that *Alice* is required to inform *Bob* of her wish to communicate in message one before interacting with the trusted third party *Steve*. In this way both peers were involved with obtaining the session key. The protocol is shown in Figure 2.5

| | | | | | |
|---|---|---|---|---|---|
| 1. | $A$ | $\rightarrow$ | $B$ | : | $N, A, B, \{\|Na, N, A, B\|\}_{K_A}$ |
| 2. | $B$ | $\rightarrow$ | $S$ | : | $N, A, B, \{\|Na, N, A, B\|\}_{K_A}, \{\|Nb, N, A, B\|\}_{K_B}$ |
| 3. | $S$ | $\rightarrow$ | $B$ | : | $N, \{\|Na, K_{AB}\|\}_{K_A}, \{\|Nb, K_{AB}\|\}_{K_B}$ |
| 4. | $B$ | $\rightarrow$ | $A$ | : | $N, \{\|Na, K_{AB}\|\}_{K_A}$ |

Figure 2.5: Otway-Rees Protocol

Now three nonces are used: one to assure *Alice* of *Bob's* identity *(Na)*, one to assure *Bob* of *Alice's* identity *(Nb)* and one to identify the run of the protocol *(N)*. In the first message, *Alice* generates the session identifier nonce $N$ and her nonce *Na*. She sends these and *Bobs* identity in the clear, plus a certificate containing all the unencrypted data, plus her identification nonce, now encrypted under the key she shares with *Steve*. Message two shows *Bob* generating his own identification nonce *(Nb)*, and generating a certificate of himself to forward *Steve* using the same syntax *Alice* used. He will then transmit the message composed by the session nonce, *Alice's* identity, his identity, the certificate from *Alice* received from message one and his own certificate encrypted with the key he shares with *Steve*.

When *Steve* receives message two, he learns from the outer unencrypted part the keys he needs to use to decrypt the certificates. He then generates a fresh session key for Alice and Bob to communicate *($K_{AB}$)*, and generates two certificates: One for *Alice* containing her identification nonce and the session key encrypted under the key he shares with *Alice*, and an equivalent certificate containing *Bob's* identification nonce and the session key, now encrypted under the key he shares with *Bob*. *Steve* then sends message three, which is composed of the session identification nonce and the two certificates. In message four, upon receipt of message three *Bob* learns the session key and forwards *Alice's* certificate together with the session identification nonce.

| | | | | | |
|---|---|---|---|---|---|
| 1. | $A$ | $\rightarrow$ | $C_B$ | : | $\{Na, A, B, \{\|Na, A, B\|\}_{K_A}\}$ |
| 1'. | $C$ | $\rightarrow$ | $A$ | : | $\{Nc, C, A, \{\|Nc, C, A\|\}_{K_C}\}$ |
| 2'. | $A$ | $\rightarrow$ | $C_S$ | : | $\{Nc, C, A, \{\|Nc, C, A\|\}_{K_c}, Na'\{\|Nc, C, A\|\}_{K_A}\}$ |
| 2". | $C_A$ | $\rightarrow$ | $S$ | : | $\{Nc, C, A, \{\|Nc, C, A\|\}_{K_c}, Na\{\|Nc, C, A\|\}_{K_A}\}$ |
| 3". | $S$ | $\rightarrow$ | $C_A$ | : | $\{Nc, \{\|Nc, K_{CA}\|\}_{K_C}, \{\|Na, K_{CA}\|\}_{K_A}\}$ |
| 4. | $C_B$ | $\rightarrow$ | $A$ | : | $\{Na, \{\|Na, K_{CA}\|\}_{K_A}\}$ |

Figure 2.6: Otway-Rees Simplified Protocol Attack

Burrows ET al. [38] later suggested the same guarantees could be achieve by a protocol that used only two nonces, and mistakenly suggested that *Nb* need not to be encrypted. Mao and Boyd [80] and later Paulson [101] found attacks in the simplified version. Paulson showed that the version that encrypted *Nb* was secure with respect to the assumptions, but found an authenticity flaw when the nonce *N* was substituted by only *Na* in the protocol. Paulson's attack on the simplified version is shown in Figure 2.6

The attack discovered by Paulson (Figure 2.6) starts when *Alice* tries to start a session with *Bob*, but the message is intercepted by *Charlie*, who learns the identification nonce for that session *(Na)*. *Charlie* will then start a new session with *Alice* as shown in message 1'. *Alice* believes *Charlie* is an honest agent and will forward the certificate provided by *Charlie* as a request to *Steve*, the trusted party, now adding a new identification nonce *(Na')*. *Charlie* again intercepts the message. *Charlie* now (2") mixes up the message *Alice* tried to send before to *Steve* and that contained *Na*, and tricks *Steve*. *Steve* is fooled (3") and sends *Charlie* the old identification nonce *(Na)*, now encrypted under the long term key *Steve* shares with *Alice*. *Charlie* now can masquerade (4) as *Bob*.

Seeing the sort of attacks in classical protocols we discussed above we can conclude that the achievement of protocol goals directly depends on the threat models as discussed before. The classical example of Lowes attack on Needham-Schroeders Public key protocol demonstrates such an important relation.

Finally the attack from Paulson on the simplified version of Otway-Rees reinforces the importance we should give to details. On the same note, Paulson shows us a very complex attack that may never be found on a trial and error basis, showing the need for thorough techniques for protocol verification. Taking this into consideration we will see in the next sections the methods developed for the formal analysis of protocols. We will try to group them and outline what they support and what their qualities and drawbacks are.

## 2.3 Formal Methods for Protocol Verification

These attacks demonstrate how tricky it is to design security protocols. They are distributed computations running in a hostile environment. One of the main innovations in computer security and security protocols in the last two decades has been the use of formal methods to address the complexity involved.

Formal methods for protocol verification normally work with abstract protocol models based on how cryptographic primitives and computer networks work. Even with a great level of abstraction to simplify the task of verifying a real world example, the problem of verifying if a security protocol provides the desired security properties is undecidable [108]. Trying to overcome this undecidability, researchers have applied a series of techniques: term rewriting, model checking, theorem proving and modal logics among other things.

During the development of such methods and tools for security protocol verification, designers are faced with a triad of limitations that need to be imposed to treat the inherently unbounded complexity of secure distributed execution.

Having this in mind we will start looking over the three classes of methods. This will help us to better understand how the methods operate and how efficient they are. We must remember that efficiency is not measured only by the triad of limitations stated before, but by a series of factors, like learning curve, automation and capacity for representation of novel problems.

An early attempt on formalising security protocols was suggested by Dolev and Yao [51] in their seminal paper, where the system was modelled as a machine used by the intruder to generate words. Their *Spy* model became later the standard into security protocol verification, but the fact that the *Spy* did not start with private information was a serious drawback. They presented several algorithms to their proposed system trying to capture certain classes of protocols, but failed in proving any authenticity and freshness, due to the fact the model was not capable of representing the storing of data by peers executing the protocols.

### 2.3.1 Belief Logics

Belief Logic is normally considered the first attempt towards the solution of the security protocols' verification problem. Burrows, Abadi and Needham [38] demonstrate the usage of a logic based on beliefs. With it we can compute what an agent would infer by the usage of formulae. BAN logic, as it became known, allowed for short and abstract proofs regarding security protocols and sparked a series of research on other more powerful methods.

BAN logic is simple. It is based on simple execution rules, it is reasonably easy to understand and it is capable of representing most protocol constructions. Although simple, BAN logic can be used to prove a variety of properties in security protocols. The fact BAN logic does not consider an intruder on the network and do consider all agents honest makes it very weak in proving any property related to confidentiality. The main weaknesses regarding BAN logic came after it was used to detect flaws in several protocols, but clearly missed some important problems. Some protocols which were claimed to be correct under BAN logic were shown to be deeply flawed [95]. To these claims the authors argued that they were violations in the basic assumptions of the logic especially that, no message should give away secret keys. Later it was highlighted by Meadows [87] that this could not be a valid assumption since it would conflict with the assumptions of the widely accepted *Dolev-Yao* threat model.

With the popularity of BAN logic a series of attempts to fix its problems were tried. Most of the propositions were by creating extensions to its initial propositions. Extensions generated some new logics, such as BGNY [35] and some automated implementation, such as Schumman's [113] using the theorem prover SETHEO. But they generally traded BAN's simplicity for coverage or suffered the same drawbacks as the original work from Burrows, Abadi and Needham. A clear advantage of using it with an automated tool, such as the one proposed by Schumman, was that when a proof attempt was not achievable, this would often indicate that an attack could be present.

### 2.3.2 State Exploration

In the State Exploration approaches we can state that a protocol is normally verified and characterised by the set of all possible traces it can yield. The idea behind these approaches is that the verification will explore all possible paths the execution of a protocol can take and assure that all conditions specified hold in each step. One clear limitation of state exploration approaches is the infinite search spaces for agents, runs and knowledge space. Principals can engage in a number of protocol runs, and simultaneously be in different roles, and the attacker can generate an infinite number of different messages.

We will see below that most state exploration techniques use theoretical results to avoid exploring the entire state space. Most of these theoretical techniques are corroborated by works like Syverson ET al. [121] which proves that, whenever there is an attack present in a protocol, there is an attack where compromised principals only sends valid protocol messages. Another work by Rusinowitch and Turani's [109] shows that whenever there is an attack in a given security protocol there is at least one of polynomial size for bounded number of sessions and players.

One of the main advantages of state exploration approaches are their expressiveness and automation. Also a lot of attention is attracted because these approaches normally detect the lack of certain properties to which they normally can construct attacks. These attacks are verifiable by the counterexamples they generate from failed traces. In the next section we will be seeing some initial attempts of state exploration approaches (§2.3.2.1) and then look deeper into the two main strains for state exploration, which are Model Checking (§2.3.2.2) and Strand Spaces when automated by computerised tools (§2.3.2.3).

#### 2.3.2.1 Early Attempts

As already stated, the earliest attempt for verifying security protocols can be credited to Dolev and Yao [51], who tried some state exploration techniques. However the approach only considered the verification of secrecy and accounted for a few cryptographic primitives. It came to be important to the field of verification of security protocols mainly by its proposed threat model.

Clearly due to its limitations it was largely ignored in terms of the verifications attempted by them and remained for over ten years under the shadows together with the whole field of security protocol verification. Although BAN-type logics were very popular in 1990s, just after 1995 that the protocol verification field came to light again, especially with efforts from Lowe, Meadows and Paulson. With the problem revived, other methods and approaches started to come to light, as we will see in the next sections.

#### 2.3.2.2 Model Checking

The successful use of model checking can be credited to Gavin Lowe [77], who approached the problem of verifying security protocols taking into considerations Hoare's calculus of Communication Sequential Processes (CSP) [68]. To verify a protocol, Lowe decided to model every entity in the communication system by creating a CSP model for the agents taking part in the protocol, the attacker using a *Dolev-Yao* fashion and the underlying communication network. He then applied a standard CSP theory of traces to conduct a deep analysis.

In his attack into Needham-Schroeder's Public Key protocol, he experimented with Failures Divergences Refinement (FDR), a model checker to verify CSP programs. Although powerful, the task of producing a CSP description for a given protocol is time consuming and requires a high level of skill. He then developed Casper [79], which translates a standardised protocol description for a security protocol into a CSP model that can be then used with FDR. His results were encouraging for many other researchers, and sparked the construction of specialised tools for security protocol verification.

Another important model checking approach is the one by Basin [16], who combines complementary aspects of model checking with those introduced by Paulson in theorem proving approaches (see Section 2.3.3). The approach consists in dealing with the state explosion problem by implementing lazy data types. By using lazy data types without evaluating their arguments, he could represent and compute terms regarding infinite data sets.

The motivation from Paulson's work [101] brings Basin to use trace-based interleaving semantics for modelling security protocols, which are formalised as infinite trees with the branches being traces. Adding a tree's child node corresponds to extending the trace, capturing then a protocol execution step or an action from the attacker. If there is an attack it is located in some node of the tree but this node can be arbitrarily deep. Finding it can be a problem in this lazy data type approach. To make the search less problematic, Basin uses some heuristics to simplify it. One is to prune traces that contain spurious events, such as messages that do not obey the rules on how the security protocol executes. Another one is to assign higher priority to events involving the start of a new execution or for an intervention from the attacker.

Basin et al. [15] evolved this approach to a tool called On-The-Fly Model-Checker (OFMC), which combines the lazy data type proposed initially with a set of symbolic techniques. This tool helps to narrow the search in a demand-driven way towards the attacker actions, which reduces the search space drastically without losing any attacks. The model that OFMC uses to describe protocols and to analyse them is based on a two level specification language similar to a byte-code execution. We have a high-level specification language called HLPSL where the main protocol descriptions are made. Then there is a low-level specification language called IF (Intermediate Format) where the verification takes place. This two level scheme was developed under the AVISPA project. The AVISPA project supports the integration of four back-end search engines: the On-The-Fly Model-Checker [15], the Constraint-Logic-Based Attack Searcher [44], the SAT-based Model-Checker [9] and the Tree Automata based of Automatic Approximations for the Analysis of Security Protocols [34]

In fact the results brought by experimenting OFMC with the AVISPA-built library of protocols [8] showed that it is capable of finding attacks present in real protocols as well as to discovering new paths to the previously known attacks [17]. It is fast in determining validity or giving a counter example for almost all protocols in the library. The main drawback of OFMC is its incapability of verifying group security protocols.

Model checkers have been very successful in the verification of security protocols despite their clear limitations on addressing the problem without limits. Their success is mostly justified by their targeting into find security bugs, which are then easily demonstrable in real world examples by following the execution path taken to achieve the inconsistency.

#### 2.3.2.3 Strand Space Models

As already stated before, the main motivation of model checking approaches is to search if in the execution space there exists a state that violates any of the established security goals of the protocols being verified. Model checkers aim mostly to find bugs, not to claim a protocol correct. Motivated by this limitations of model checking, Fabrega, Herzog and Guttman [53] proposed the strand space model. In the strand space model, a strand is a sequence of events an agent can execute within the protocol description and a strand space is the set of strands that represent all the behaviour present within the protocol, both honest agents and the attacker. As strand spaces were designed as a "hand-proof" method some would argue that they are no state exploration tools, but when automated this method behaves similarly to state exploration ones.

A strand space is a graphical collection of strands that represent the relations between agents during protocol execution. The strand is composed of events that can be either the sending or receiving of messages. These are graphically represented as nodes. In a strand space model, we have two types of strands, those of honest agents containing the agent's actions in one particular run of the protocol, and those from the attacker representing all the actions he is capable of doing. The attacker has capabilities defined by the threat model such as interceptions, fabrications and rearrangement message parts. If agents are involved in several parallel runs, each one will generate a new strand. Attacker's actions can be created by the connection of many different strands. The verification of a strand space model also has the notion of bundle. A bundle is a finite acyclic sub graph of the strand space, to which properties are proven upon using induction. These proofs were originally carried out by pen and paper, which made it very demanding.

Song [116] created an automated tool for automatically checking security protocols, which is called Athena. Athena uses techniques from both model-checking and theorem proving to extend the strand space model so that it is both automatic and able to prove correctness of security protocols with arbitrary numbers of agents and runs. Athena includes optimisations to improve efficiency, such as the use of a pruning procedure based on novel theorems and an automatic evaluator for well-formed formulas. Although promising, Athena does not have guaranteed termination, which is only achievable by limiting the size of message terms or the number of parallel runs.

Strand spaces provide a simple and graphic way of expressing the relationship between different parts of a protocol and can produce efficient verification by the usage of automated proof techniques. The main drawbacks of strand space models are inflexibility, especially if compared with Paulson's inductive method [101] and the necessity of imposing limitations in order to speed up the automated verification process and to guarantee termination. Also, it cannot generate attack counter examples.

The field of state exploration is probably one of the most active in verifying properties in security protocols. As seen above, we have different techniques that normally work with a goal of detecting protocol flaws, aiming to explore the execution space in them. We will see below a different approach, not based on exploring states, but into constructing logical proofs regarding security protocol's properties.

### 2.3.3 Theorem Proving

The main goal of theorem proving techniques is to produce formal proofs of a given specification, based on a set of logical axioms and a set of inference rules. Theorem proving approaches can be divided in to two main groups: Interactive (§2.3.3.1) approaches, typically based on Higher-Order Logic , which can inductively simulate an infinite number of agents and protocol sessions, but requires user guidance for achieving the proofs, and automatic approaches, based on First-Order Logic (FOL) (§2.3.3.2), which sacrifice some of the expressiveness of types and quantification in order to acquire automation, making them attractive to quickly generate attacks, or counter-examples, for the properties being verified.

#### 2.3.3.1 Higher-Order Logic

The verification of security protocols using theorem provers in Higher-Order Logic is due to Paulson [101]. He introduced the inductive method of protocol verification where protocols are formalised in typed higher-order logic as being an inductively defined set of all possible execution traces. An execution trace is a list of all possible events in a given protocol. Events can be described as the sending or receiving of messages, as well as off-protocol gathered knowledge. The attacker is specified following *Dolev-Yao's* propositions. The attacker, or Spy as it is represented in the method, has his knowledge derived and extended by two operators called *synth* and *analz*. Operator *analz* represents all the individual terms that the attacker is able to learn using his capabilities defined by the threat model within the protocol model, and *synth* represents all the messages he can compose with the set of knowledge he possesses.

Protocols are defined as inductive sets constructed over abstract definitions of the network and cryptographic primitives. Proofs about protocol's properties are written as lemmas. Lemmas are constructed taking what properties we desire to achieve within the set of all admissible traces, and are typically proven inductively. This framework is built over induction, which makes the model and all its verifications potentially infinite, giving us a broad coverage and a lot of flexibility. This approach was already used to prove a series of classical protocols [99, 101] as well as some well-known industry grade protocols, such as the SET online payment protocol, Kerberos and SSL/TLS [26, 23].

Some of the criticisms about Paulson's Inductive method concern the difficulty in achieving proofs, the lack of automation and also the very steep learning curve. But Paulson's formalism is very expressive and capable of capturing a good model for most security protocols. It is especially appealing because it deals naturally with arbitrary number of agents, parallel runs and components in the knowledge set of agents.

As this approach is the focus of this thesis, we will describe it in a much deeper way. We will describe the method in Section 2.4 and the verification process in Subsection 2.4.2. The justification for its choice is three-fold: The method is malleable enough to enable experimentation with novel concepts and doesn't suffer the triad of limitations explained above. Finally the main criticism it faces was easy to overcome since we have local expertise available to guide us.

#### 2.3.3.2 First-Order Logic

One of the the first attempts of using automated theorem proving to verify security protocols properties was performed by Meadows [87] with the NRL Protocol Analyser. The NRL Protocol Analyser is a tool built on *Prolog*, specially tailored to verify security protocol properties. The *NPA*, as it became known, aims to prove that a number of protocol states are reachable and by that to satisfy authentication and key distribution properties for a given principal. *NPA* was used to find several attacks into protocols, such as in Simmon's Selective Broadcast Protocol [115], Neuman-Stubblebine Re-authentication protocol [96] and Aziz-Diffie Wireless Communication protocol [13].

Although NPA is based on a subset of first-order logic it is very interactive, which makes it as complex to use as an interactive theorem prover. Also to be able to gather significant results it requires a savvy user to guide the tool in the search for a proof. Other problems regard the non guaranteed termination and its incapability of converting a protocol description into a set of term-rewriting rules.

Weidenbach [124] developed a method based on Paulson's Inductive Method that reduces the problems from higher-order logic to first-order logic. He keeps the same inductive representation proposed by Paulson but gains the automation from first-order theorem proving tools. Weidenbach's approach is based on *monadic Horn clauses*. *Monadic Horn clauses* are a fragment of first-order logic that is expressive enough to allow infinite inductive models, but contain at most one positive literal. All the predicates take only one argument. The method is implemented by first converting the protocol descriptions and its associate requirements into first-order monadic Horn formulae. These formulae are passed to SPASS, a first-order logic theorem prover that will saturate the results trying to prove any flaws. The outcomes of a verification using Weidenbach's method either terminates if the saturation happens meaning the protocol properties and description given to SPASS yield no security breach, or it terminates with a counterexample. These counterexamples show the saturation failure and normally consists of a valid non minimal attack path.

The method developed by Weidenbach has successfully been used to find new attacks in protocols, including the Neumann-Stubbline protocol [124]. We discovered and attack of our own within the Brazilian Electronic Bill of Sale protocol [104, 82], which we examined during some initial experimentations with inductive modelling for security protocols. Our remark from this experimentation is that Weidenbach's method can be seen as a reasonable teaching tool to Paulson's Inductive Method. For novice users it can teach the basis of the inductive verification process due to its very close relation, but without suffering from some major drawbacks regarding the strictness of the higher-order description and the lack of automation. The criticism for Weidenbach's method come from his original attacker model being weaker than the standard *Dolev-Yao*, but this is easy to overcome as we noted in our analysis [82]. A second criticism is that it is not as expressive as Paulson's inductive method because it does not enforce types and does not provide quantification. It is not clear for example if the approach used for modelling time-stamps in Paulson's inductive method would work in Weidenbach's model.

Blanchet [31] developed an approach based on Horn clauses for verifying secrecy properties in security protocols. It takes a specification using spi-calculus [2] which is automatically translated into a set of Horn clauses. These Horn clauses are then passed to a first-order prover that will assert it's properties. Similarly to Weidenbach's approach, Blanchet can also generate a counter example which can be used to recreate the steps needed to achieve a successful attack within the real protocol. Blanchet's innovation concerns the input using spi-calculus and the way he treats freshness within protocol messages.

The method is mainly focused on secrecy and authentication, but we have seen cases where properties like privacy, traceability and verifiability were also considered. The main advantages of Proverif [31], the tool Blanchet implemented, is its capability of working with unbounded numbers of parallel sessions, the safety of always finding a violation of secrecy and its automation and ease of use. The main drawbacks of Blanchet's method are its assumption of a bounded number of agents, the yielding of false positives especially due to how freshness is modelled, and its incapability of counting messages on channels, what makes extremely difficult to use it for time-stamp based protocols.

Another first-order logic method is by Cohen [46]. Cohen's method is based on the verification of invariants using first-order logic with the formalisation of the protocol as a transitions system. Each state represents the set of transitions that have been executed and the set messages that had been sent onto the network. The system then constructs first-order invariants capturing the properties we want to assert. One of the main characteristics of the method's implementation in *TAPS* [46] is its capacity to generating a number of protocol invariants from a protocol description. These invariants are then used to establish the security goals automatically. *TAPS* has been used to verify a large number of security protocols [46], and its small user interaction is one of its main advantages, especially if compared to similar methods focused on proving invariants, such as Paulson's. Critics on this method normally point to its inability of generating counterexamples, or attacks. Although controversial, some critics say that, by not requiring user input, it undermines the understanding of flaws the protocol can have as it does not add to the learning of the process as other tools do.

Finally, we come to Graham Steel's method, another first order variation of Paulson's inductive method. Steel's approach is targeted to address the verification of group protocols. This method is focused into capturing the verification of group protocols where an arbitrary number of peers may be involved in a single protocol run, as defined by Meadows [88]. He proposes a scheme based on proof by consistency aided by a first-order logical framework.

The method proposed by Steel, being a first-order variant of Paulson's, specifies a protocol as an inductive data type, which captures all possible traces. Axioms specify how the trace will be expanded, either by agents following the protocol or by the Spy. The model captures the possibility of having an undetermined number of agents to get involved in one or more protocol runs. They may play any role and are able to issue an arbitrary number of fresh components. The Spy and the protocol are specified as Horn clauses and the negated property is passed to the first-order theorem prover. The output is either a counterexample or saturation in a similar fashion to Weidenbach's approach. The focus of Steel's approach is finding bugs or attacks.

There are some similarities between Steel's verification objectives and mine. To clarify the similarities and differences, Steel is focused on group protocols where an arbitrary number of participants may be involved in a single run, but still under the same message framework of Unicast. Our work in different in the way it implements capabilities for other types of message casting techniques and we are focused on the original ideas of Paulson's inductive method. Also our main focus is to prove correctness rather than finding attacks.

Having reviewed the main methods in protocol verification up to date in the Unicast world, we will cover now an in-depth overview of Paulson's inductive method. His method is the basis to most of approaches based on theorem proving and is one of the few that have no restrictions on the triad of limitations we stated above. It is able to verify properties in protocols with and unbounded number of player and sessions, as well as an unlimited knowledge set for the attacker. In the next section we will be looking for its construction and the verification process it demands. We will also briefly look over the principle of Goal Availability, which generally shapes the statement of properties for verification.

## 2.4 In-depth view of the Inductive Method

Paulson's inductive method is the approach of choice for this thesis. It suffers the fewest drawbacks while addressing the triad of complexities we stated in Section 2.3. We can also cite the extensibility of the method, the influence the method imposes into other methods as seen in the above section, and the availability of local expertise to help us through the process.

The idea of this section is to bring to the reader an up-to-date summary of how the inductive method was at the beginning of the developments that lead to this thesis. We don't want to be complete in our coverage, but complementary to other descriptions [101, 20]. The text for the next sections will have its structure heavily based on Bella's [20] own review for his book which takes into consideration the method as it was at the beginning of his proposed extensions. We will try to create an updated summary so that the reader can understand the basis over which we built our approach to address multicast protocol verification. We need to start giving the due credit to Paulson and Bella for all the ideas below. For a complete understanding of the mechanics behind the Inductive Method we recommend reading Isabelle/HOL tutorial and chapter 3 of Bella's book into formal correctness of security protocols [20], as they produce deeper views into the method's basis.

Paraphrasing Bella [20], we start by stating that Paulson's Inductive Method, or just the Inductive Method, is a natural way of constructing a process to help with the verification of security protocols. This will inherently happen while the protocol is being extended by parallel and potentially infinite executions and in the presence of an attacker. The inductive method [101] addresses the security protocol verification problem in a completely unbounded fashion. It does so by representing the protocol models and operational functions using structural induction, where properties are asserted inductively and verified by inductive proofs over a completely inductive model.

We can split the verification process using the Inductive Method into two phases: The first phase concerns the protocol specification and the infrastructure representation which is needed for representing protocol's execution for the purpose of verification. The second, called properties verification phase, proofs are built for the desired properties yielded by the protocols over the specification.

Induction in the specification phase defines the interactions a protocol can create to generate all the possible extensions achievable by protocol execution. As an inductive set definition, it takes into consideration all the actions the involved peers including the attacker can take during protocol execution. The execution is not bounded in the number of parallel sessions, the number of agents or size of the knowledge set for each agent. The verification phase consists of translating the security requirements into lemmas or theorems that can be proven by induction. Actions are not mandated to happen due to the structural induction nature of the specification. The method is mechanised using the generic theorem prover Isabelle/HOL [97]. This is not a requirement, since the method is tool independent, and could be done even with pen and paper.

Isabelle/HOL is an interactive theorem prover which can be used to assist a great range of formalisation efforts. It requires user guidance to achieve formal proofs of the specified formalisation goals. Its support for higher-order logic allows it to quantify over functions, predicates and sets. Isabelle/HOL is equipped with a set of logical resolution tools that assist the user into demonstrating proofs with speed, consistency and accuracy. Among these automations, we must cite its simplifier, which implements term rewriting and arithmetic resolution, and its classical reasoner, which implement automated methods for logical resolution.

In the last years, Isabelle/HOL has been heavily updated and integrated with other theorem provers to assist it with even more automation. We must cite especially the existence of Sledgehammer [89], which is a resolution tool in Isabelle that converts the problems being analysed in Isabelle/HOL into first-order logic and passes the goal analysed to be automatically proved using a FOL automatic theorem prover. In case of success, the proof is reconstructed in Isabelle automatically and accounted for the higher-order resolution. Its consistency is checked using the higher-order logic framework.

In Isabelle/HOL, we start our formalisation with abstract definitions of our problems and properties. We follow with the verification of our properties by stating theorems, which are demonstrated using proof commands available with Isabelle. Our proof commands can be direct calls to the simplifier or to the classical reasoner, where we guide the tool with our proof sketch. As already noted by Bella [20], this process is difficult and demands a great effort in terms of user interaction with the tool. But, despite these drawbacks the difficulty of creating a sound proof in Isabelle/HOL demands from the user a deep understanding of the properties being proven. This normally will increase the confidence that the process is correct. In our case of security protocol verification, this is an important factor in favour of the inductive method: it not only is able to prove a protocol correct but also demand a deep understanding of what has been done.

Isabelle/HOL being a generic tool for specification and verification, which includes a repository of already verified theories. Theories are files where we find a specification followed by proof scripts that demonstrate properties present in them. Theories can be reused by an inheritance scheme and their development can be absorbed into extending or specialising a specification into the properties we want to demonstrate. Most theories we see in Isabelle/HOL inherit the specification of the Higher-Order Logic system Isabelle implements. The Isabelle/HOL distribution library has formalisations on a variety of fields, such as lambda-calculus, Java source and byte-code and aspects of UNIX file system security. This gives us ready-to-use tools in specifications and verifications we want to conceive.

We will see bellow how the inductive method is implemented and how it is mechanised using Isabelle/HOL. We will cover all its construction and the specification and verification process for security protocols. We will try to cover the description using examples of protocols verified with the inductive method.

### 2.4.1 Theories and Definitions

The theories implementing the Inductive Method and are structured into different files according to what is being specified and verified. The method starts with the theory *Message.thy*, which specifies how the messages exchanged in security protocols are constructed, as well as the main operators we use within the method. *Message.thy* inherits its properties from the Main theory which implements the specification of the Higher-Order formalism in Isabelle/HOL. Other important theories are *Event.thy*, which inherits from *Message.thy* and accounts for the specification of the communication layer with the sending, receiving and noting events for example. We also have *Public.thy* which inherits from *Event.thy*, which despite the narrowly chosen name, accounts for the specification of symmetric and asymmetric cryptographic primitives.

These three theories are the core of the Inductive Method within Isabelle/HOL. But we must also note that specialised theory variants exist in the method. Bella's smart-card verification [20] produced a family of theories that inherit from *Message.thy* and implement the intrinsic of smart-card events (*EventSC.thy*) taking into consideration the existence of channels between the cardholder and the smart-card for example. Theory *Smartcard.thy*, inherits from *EventSC.thy*, and account for the characteristics of some cryptographic keys specific to smart-cards protocols' modelling. Our work in this thesis will similarly create a new family of theories for the verification of non Unicast-only security protocols. Theory *EventMC.thy* inherits from *Message.thy* and for threshold cryptographic primitives *PublicSS.thy* inheriting from *EventMC.thy*.

Security protocols are specified using the infrastructure described above. We will choose a theory corresponding to the cryptographic implementation and communication scheme we need. From that point we will start the description of our protocol using the syntax that we will define below.

We will not follow the exact structure of the theory files, but will try to rearrange the descriptions in a more didactic way. The sections below will show the definitions for the data types present in the method. We will also discuss the implementation of the threat model, the operators, how a classical protocol is modelled and the idea of trace. After that we will discuss the verification phase (§2.4.2) passing before through the principle of Goal Availability (§2.4.2.1).

#### 2.4.1.1 Agents

As noted early by Paulson [101], the modelling of an unlimited-size population of agents is essential. The implementation is done by establishing a bijection between the population of agents and the set of natural numbers. Agents definition is introduced as shown in Definition 1.

**Definition 1.** *Agent datatype definition*

```
datatype
  agent = Friend nat
        | Server
        | Spy
```

Definition 1 shows the categorisation of agents into three distinct classes. First we have the definition of a friendly agent by the bijection explained above. This implies that for any given natural number there exists a related agent in the system. The second category regards the trusted third party normally used in symmetric key protocols to which we share our long-term keys. Finally we have the attacker, which is categorised separately. The Spy, how the attacker is named in the inductive method, is here represented as a single entity which shows a clear derivation of the all-powerful omnipresent attacker model from the *Dolev-Yao* threat model [51].

As noted by Bella [20], this scenario captures classical protocol specifications. In his verification of the Kerberos family of protocols, he did experiment with extending the data type but eventually modelled his extra trusted servers as normal agents that could not be corrupted by assumption. In fact his approach suggests that the definition of agents in the inductive model could only need the differentiation from agents following the protocol and the Spy. This would incur in an initialisation overhead when using trusted third parties to give them the long term keys knowledge and the refinement of lemmas to cope with this new specification, but would enable us to easily generalise the model.

The specification of the Spy makes sense when regarding the *Dolev-Yao* model, which is a standard attacker model for protocol verification nowadays. The existence of a single attacker entity denotes this clear relation between the method and the *Dolev-Yao* attacker.

#### 2.4.1.2 Cryptographic Keys

The specification of cryptographic keys in the Inductive Method starts with the introduction of a free type *key* as a derivation of the type *nat*. We start specifying cryptographic keys with the definition for long term keys usually shared with a trusted third party in symmetric cryptography based protocols. The specification of the function *shrK* is shown on Definition 2.

**Definition 2.** *shrK function definition*

```
consts
  shrK ::  "agent => key"
specification (shrK)
  inj_shrK: "inj shrK"
```

As we can see in Definition 2, a shared key is specified as an injective function taking an agent and returning a key. If both keys are equal we call the cryptography symmetric, if they are different we call the cryptography asymmetric as we already covered in this thesis. To specify this operation we have the declaration of the function *inky* as shown in Definition 3.

**Definition 3.** *invKey function definition*

```
consts
  invKey ::  "key => key"
specification (invKey)
  invKey [simp]:  "invKey (invKey K) = K"
  invKey_symmetric:  "all_symmetric --> invKey = id"
```

The function *invKey* from Definition 3 is a function from data type key to key specified by two rules. The first rule is a simplification one that says that the double application of the function brings us back to the original value. The second rule establishes true if the function *invKey* is equal to its identity.

By having an inversibility function for keys, we can establish a set that will help us to distinguish between the different categories of keys. We will define a key set for all symmetric keys calling it *symKeys*. Definition 4 show its specification.

**Definition 4.** *symKeys set definition*

```
constdefs
  symKeys ::  "key set"
  "symKeys == K. invKey K = K"
```

The symmetric key set is defined as containing all keys where the inverse of the key by the application of the function *invKey* is itself. This set helps us to identify the symmetric and asymmetric key being used. But it is also important to help identifying the session keys being used in the protocols we are verifying. By stating$K \in$ *symKeys* $\wedge$ $K \notin$ *range shrK* we can create a second class of symmetric keys that are not long-term so that it represents sessions keys in our protocols being verified. We will use a similar approach later in our definition of threshold cryptography.

To establish that our long-term keys are symmetric we assert an axiom.

**Definition 5.** *Axiom for symmetric usage of shared keys*

```
axioms
  sym_shrK [iff]:  "shrK X ∈ symKeys"
```

We define asymmetric keys in term of the public keys since they are available for all agents during a protocol run. We will derive the notion of private keys by the usage of the function *invKey*. Definition 6 shows us the specification for the *publicKey* function.

**Definition 6.** *publicKey definition*

```
datatype
  keymode = Signature | Encryption

consts
  publicKey ::  "[keymode,agent] => key"

specification (publicKey)
  injective_publicKey:
   "publicKey b A = publicKey c A' ==> b=c & A=A'"
```

The *publicKey* function is defined as from a pair *keymode* and *agent* into a *key*. The data type *keymode* is defined as having the values for Signature and Encryption. This is done so that we can differentiate later during protocol verification the intended use for an asymmetric key. The specification of the *publicKey* function is done by an injective definition that if the public key for an agent in a key mode is equal to a public key of another agent in a different key mode, this implies that both agents and modes are the same. This is done so that we can trust no agent will have a key equal to some other agent's.

Another important assertion is that no private key is equal to a public key for an agent. This is done by the axiom shown in Definition 7.

**Definition 7.** *privateKey axiom definition*

```
axioms
privateKey_neq_publicKey [iff]:
  "privateKey b A ≠ publicKey c A'"
```

To make our public key cryptography specification more usable in the way we implemented it, and for the sake of minimising the effort on using the specification we define some abbreviations. The abbreviations help us so that we can refer to our asymmetric keys in an easier way while still using the full extension of the specification.

We have the abbreviations for *PubEK* and *PubSK*, which expand to public encryption and signature keys respectively. We also have the abbreviation for *privateKey* which inverts the key of an agent in the desired key mode. Then we have the abbreviations for *priEK* and *priSK*, which represent the private keys for encryption and signature. Finally, when we refer to a public (*pubK*) or private (*priK*) key without referring to their key operation mode, we will be referring to encryption by default.

#### 2.4.1.3 Compromised Agents

To be able to implement a threat model within the inductive method we need to introduce support for the identification of corrupted principals. The inductive method comes with two major representations for compromised agents. Definition 8 shows us the specification of the set of *bad* agents.

**Definition 8.** *bad set definition*

```
consts
  bad :: "agent set"
specification (bad)
  Spy_in_bad [iff]: "Spy ∈ bad"
  Server_not_bad [iff]: "Server ∉ bad"
```

The set of *bad* agents is specified by the explicit inclusion of the *Spy*, and by the explicit exclusion of the trusted third party the *Server* in this case. During the application of the method we can explicitly assert the membership or not for the agents running the protocol. We will see later that an agent belonging to the set *bad* means that he shares his long-term secrets with the *Spy*. Sharing long-term secrets with the *Spy* does not only mean to give away secrets at start, but also to give access to the *Spy* to other secrets learnt during protocol execution.

Although this representation seems enough, Bella [20] noted that some refinement was required for the verification of accountability protocols. As we will see next (§2.4.1.6), the membership of an agent to the set *bad* automatically gives the Spy access to all this private information, including the agent's long-term shared and private keys. The Spy than can use such keys to acquire knowledge and participate in the protocols impersonating the bad agent.

A subtle change is necessary to represent the non-repudiation required for accountability protocols. Paraphrasing Bella [20] once more, the signature of compromised agents is worthless, but the Spy's own signature is still valid when he is acting as himself. For encompassing such subtlety, Bella specified the set *broken*, as shown on Definition 9.

**Definition 9.** *broken set definition*

```
constdefs
  broken :: "agent set"
  "broken == bad - {Spy}
```

If an agent is in the set *broken* the Spy has access to his keys and confidential material, but clearly this agent is not the Spy. By having this refinement we can create assertions for our goals where the Spy is only acting as himself, although he has access to other peers' confidential material. Thus we can state the lack of trust peers may have on each other. We also used *broken* to specify the compromised servers in the Frank-Reiter sealed-bid auction protocol.

#### 2.4.1.4 Messages

Pragmatically defining, messages are the composite payload events will carry.

**Definition 10.** *msg datatype definition*

```
datatype
  msg = Agent agent
        | Number nat
        | Nonce nat
        | Key key
        | MPair msg msg
        | Hash msg
        | Crypt key msg
```

The data type *msg* from Definition 10 introduces seven different constructors for messages' payloads. We have the constructor for a *Agent* identification that requires a variable of the type *agent* which we defined in section 2.4.1.1. We have a constructor *Number* and a constructor *Nonce*, both taking natural numbers as variables. The constructor for *Number*, which was introduced by Bella for the modelling of timestamps, represents guessable numbers in the protocols. The constructor *Nonce* represents the unguessable numbers. Constructor *Key* takes a variable key, which was defined in section 2.4.1.2 as a free type.

The recursive *MPair* constructor takes two messages as parameters and will concatenate them into one message. To help the specification of complex protocols, the *MPair* constructor has an annotated syntax. A pairing of two messages is done by enclosing the message in fat brackets, like in "$\{|x, y|\}$". We also have a commutative rule that is defined by "$\{|x, y, z|\}$" == "$\{|x, \{|y, z|\}|\}$".

The other two constructors in data type *msg*, regard the presentation of basic cryptographic operations. Before describing them, we must stress that the representation of cryptography in the inductive method is perfect. We treat cryptography as fully operational black box. The constructor *Hash* represents the usage of one-way functions we may have in our protocol specification. But its abstract definition is achieved only by the method's operators (§2.4.1.8).

The final constructor is *Crypt*, which takes as variables a key and a message. It can be used to represent symmetric and asymmetric encryption, as well as to represent digital signatures. Its definition is abstractly done in the description of the method's operators (§2.4.1.8) similarly to the *Hash* constructor. The existence of the *Crypt* constructor will limit the operations an agent can perform based on the set of keys he knows.

#### 2.4.1.5 Events

An event is an action from an agent that will inherently change the state of his knowledge or that of other agents. Originally the definition from the data type *event* accounted for two types of events, the sending of a message by one agent to another agent (*Says*), and the capacity of an agent of taking notes of knowledge acquired off-protocol either by hints or by calculation is expressed by the constructor *Notes*. We can see the actual definition for the data type *events* on Definition 11.

**Definition 11.** *event datatype definition*

**datatype**
```
  event = Says agent agent msg
        | Gets agent msg
        | Notes agent msg
```

We also see on Definition 11 the existence of the constructor *Gets*, an introduction by Bella [20] to account for message reception. During his work Bella realised that guarantees should be available for agents to derive them from available facts. This required the refinement of the data type *event* to account for lost messages and for their reception.

Bella also did the verification of smart-card protocols that required the extension of the data type event to account to other subtleties of events happening in this specific domain. To help us demonstrate how easy it is to create a parallel strain for the inductive method we will show Definition 12 for the data type *events* for smart-cards.

**Definition 12.** *eventSC datatype definition*

**datatype**
```
  event = Says agent agent msg
        | Gets agent msg
        | Notes agent msg
        | Inputs agent card msg
        | C_Gets card msg
        | Outpts card agent msg
        | A_Gets agent msg
```

The changes in the data type *events* to account for smart-card protocols takes into consideration another channel of communication: the one between the card and its owner. Four new constructors were included, one sending and one receiving each side of the duplex communication. Constructor *Inputs* accounts for the channel from the Agent to the card, constructor *C_Gets* accounts for card reception from Agent to cad. Constructor *Outpts* accounts for messages from the card to the agent owning it, and constructor *A_Gets* accounts for reception from the agent for a message casted by the card in this channel.

This extension by Bella was extremely useful to our work presented later (§3.2), especially showing the potential of capturing new details when changing the event model.

#### 2.4.1.6 Threat Model Support and Initial Knowledge

To be able to support threat modelling in the inductive method we need to define the threat model we adhering to. The inductive method is originally based on the *Dolev-Yao* threat model [51]. As we already discussed (§2.2.2), the *Dolev-Yao* threat model accounts for a standard attacker that oversees the network and can selectively interfere with it.

1. The *Spy* is a legitimate agent

2. The *Spy* controls the network traffic

3. The *Spy* can perform any message operation except cryptanalysis

We see the *Spy* as an internal user of our security protocol trying to act maliciously. This is what enables Lowe's attack on the Needham-Schroeder Public Key Protocol and is a widely accepted premise when designing security protocols. This characteristic of the threat model is supported by the way we formalise agents (§2.4.1.1) and cryptographic keys (§2.4.1.2), followed by the way we introduce the specification for security protocols in section 2.4.1.9.

To be able to address characteristic number two, we need to add support for the agents to store their knowledge gathered at start and during the execution of the protocol (§2.4.1.7). This is especially true for the *Spy*. For that, we need to specify what each agent starts with in a protocol run. For that we introduce the function *initState* as shown on Definition 13.

**Definition 13.** *initState definition*

**consts**
```
  initState ::  "agent => msg set"
```

Although for the threat model *per se* the only important definition is the *Spy's* we will define it for all agents. The idea of the function *initState* is to receive an agent and return a message set that corresponds to his initial knowledge prior to the run of the protocols. To be able to do that, we need to specify what knowledge each agent starts with.

The *Server* starts any protocol runs knowing all long-term shared keys present on the environment, through the image over the range of the function *shrK*. This is then unified with his own private keys for encryption and signature. Finally we unify the previous output with the range of all public keys available, either the ones for encryption and for signature verification. With that the agent *Server* has his initial knowledge composed.

An agent in the agent population starts the protocol run with knowledge of his own private keys for encryption and signature, as well as the long-term shared keys he has with the *Server*. To that set we unify the range of all public keys available, either the ones for encryption and for signature verification from all other agents, *Spy* and *Server* included.

As we introduced before (§2.4.1.3) we can identify the compromised agents by their membership to the *bad* set. As this membership is static so that an agent cannot move between being compromised and not compromised, we can propose the specification of the *Spy* initial knowledge set as shown on Definition 14.

**Definition 14.** *Spy agent initial knowledge definition*

**primrec**
```
  initState_Spy:
    "initState Spy =
    (Key ‘ invKey ‘ pubEK ‘ bad) ∪
    (Key ‘ invKey ‘ pubSK ‘ bad) ∪
    (Key ‘ shrK ‘ bad) ∪
    (Key ‘ range pubEK) ∪ (Key ‘ range pubSK)"
```

The *Spy* starts the protocol run knowing all secrets of agents belonging to the set bad. The initial knowledge is represented by the unifications of the images of the inverses of the public keys and the long-term shared keys for the *bad* set. This is then unified to the range of all public keys available as we did for other agents. We do not need to give explicit knowledge to the *Spy* for his own keys, since that, by the definition of the set *bad*, the *Spy* is static a member.

To be able to cover the second characteristic of the Dolev-Yao threat model, we need to account for the knowledge representation and extension for the *Spy* which will be shown in Section 2.4.1.7. To address the third characteristic of the threat model, we need to enable the *Spy* to analyse the traffic. This is shown in section 2.4.1.8.

#### 2.4.1.7 Knowledge Representation and Extension

Bella proposed the extension of the method to account for each peers dynamic knowledge gathering. This is represented by the function knows, as shown in Definition 15.

**Definition 15.** *knows function definition*

```
consts
  knows ::  "agent => event list => msg set"

primrec
  knows_Nil:  "knows A [] = initState A"
  knows_Cons:
    "knows A (ev # evs) =
    (if A = Spy then
    (case ev of
      Says A' B X => insert X (knows Spy evs)
      | Gets A' X => knows Spy evs
      | Notes A' X =>
        if A' ∈ bad then insert X (knows Spy evs)
          else knows Spy evs)
    else
    (case ev of
      Says A' B X =>
        if A'=A then insert X (knows A evs) else knows A evs
      | Gets A' X =>
        if A'=A then insert X (knows A evs) else knows A evs
      | Notes A' X =>
        if A'=A then insert X (knows A evs) else knows A evs))"
```

As per Definition 15 the function *knows* takes an agent and an event list as input and returns a message set. If the event list is not empty, we then take the first element of the list, and check if the agent involved in the event is the *Spy* or not. If the agent is the *Spy*, in the case of an event *Says*, the *Spy* inserts the payload into his *knows* set. If the event is a *Gets*, the function returns without extending the knowledge set for the *Spy*. If we have an event of the type *Notes*, we check if the agent noting the message is compromised. If yes, we add the noted event into the *Spy's knows* set. If the agent in the event we are dealing with is not the *Spy*, we will extend its knowledge if and only if the agent is participating in the event in the appropriate manner.

With the specification of knowledge sets we can address fully the characteristic two for the *Spy* of the *Dolev-Yao* threat model. In fact we can, by the inspection of the knowledge the *Spy* acquired during the protocol run, represent the knowledge gathered from events that happened in the protocol run from the point of view of an external observer. As Bella pointed out [20], the *knows* function also represents each peer's point of view of the execution. This will help us also with the granularity needed to specify goals that can be achieved based on the knowledge an agent acquired during the execution of the protocol, fulfilling the principle of goal availability.

#### 2.4.1.8 Operators

To address the third characteristic of the *Spy* in the *Dolev-Yao* threat model, we need to enable the *Spy* to perform all the operations he can by the usage of knowledge he had initially or that he acquired during the protocol runs. To be able to do that we need to specify an inductive relation that enables agents to inspect the trace by breaking up messages into their atomic components, an inductive relation that enable the agents to have access to the information under the cryptographic restrictions we imposed, an inductive relation that enables all the possible combinations of messages in his knowledge to be re-composed, so that the *Spy* can attack the model with his full capabilities, and a function that establishes freshness for components present in the trace of events. These are the operators of the inductive method: *parts*, *analz*, *synth*, and *used*.

The inductive function that is capable of disassembling composed messages into its atomic forms is called *parts* and is specified in Definition 16.

**Definition 16.** *parts inductive set definition*

```
inductive_set
  parts ::  "msg set => msg set"
  for H ::  "msg set"
  where
      Inj [intro]:  "X ∈ H ==> X ∈ parts H"
      | Fst:  "{|X,Y|} ∈ parts H ==> X ∈ parts H"
      | Snd:  "{|X,Y|} ∈ parts H ==> Y ∈ parts H"
      | Body:  "Crypt K X ∈ parts H ==> X ∈ parts H"
```

It enables us to inspect the knowledge of each player for any atomic message component that we need to assert in our verification goals, independent of cryptographic restrictions. The function *parts* takes as argument a message set and returning another message set which includes the break down of all messages for a specific knowledge set to their atomic parts.

The first step of the specification from Definition 16 takes into account the injectiveness of a knowledge set into a *parts* set for that knowledge set. The second and third steps of the specification takes into account the breakdown of composed messages. In fact it disassembles the *MPair* constructor we used to compose the message to get the atomic message components. Finally, the last part of the specification recovers the encrypted components into their original form without respecting the cryptographic terms. We don't treat the break down of other constructors in the data type msg because they take only one argument and are atomic.

The function *parts* enables us to reason about traces of events without cryptographic boundaries. This type of inspection helps us specify lemmas regarding protocol wide laws, such as Regularity lemmas (§2.4.2.2), or to establish the Unicity (§2.4.2.5) and Authenticity (§2.4.2.4) properties of certain message components in the trace, regardless of point of view.

To specify the requirement three of the *Spy* under the *Dolev-Yao* threat model we have the function *analz*. Similarly to the inductive relation *parts*, it is defined taking as argument a message set and returning another message set now including the break down of messages for a specific knowledge set. The main difference of *analz* when compared to *parts* is the cryptographic restriction that is imposed on it.

**Definition 17.** *analz inductive set definition*

```
inductive_set
  analz ::  "msg set => msg set"
  for H ::  "msg set"
  where
      Inj [intro,simp] :  "X ∈ H ==> X ∈ analz H"
      | Fst:  "{|X,Y|} ∈ analz H ==> X ∈ analz H"
      | Snd:  "{|X,Y|} ∈ analz H ==> Y ∈ analz H"
      | Decrypt [dest]:
         "[|Crypt K X ∈ analz H; Key(invKey K): analz H|] ==>
        X ∈ analz H"
```

The last part of the specification decrypts the encrypted components with the cryptographic restriction as for the threat model. Here we establish preconditions for the extension of the *analz* set. These conditions are that we have a message X encrypted with key K on the *analz* set, and we also have the inverse of key K in the same *analz* set, then we can expand the *analz* set with X.

The function *analz* enables us to specify confidentiality (§2.4.2.6). By sequence it also enables any other property that depends on confidentiality.

Although the inductive relation *analz* seems straightforward, its specification regarding the existence of keys in the inductive set itself creates one of the biggest burdens when establishing proofs that mention the relation *analz* . A usual way for sorting out such added complexity is to eliminate *analz* using *analz* $H \subseteq$ *parts* $H$. Sometimes this step is not helpful and for that, in our experiments we had to come with a way of dividing the *analz* set into two parts: one containing key related material and one not containing key related material, so that we can use the above fact. This will be shown in detail on section 5.3.

Once the *Spy* has gathered and derived all possible knowledge he could from inspecting the traffic from the network, he needs to be able to rearrange those broken-down components into new messages to probe the protocol This is done by the inductive function *synth* as specified in Definition 18.

**Definition 18.** *synth inductive set definition*

```
inductive_set
  synth ::  "msg set => msg set"
  for H ::  "msg set"
  where
      Inj [intro]:  "X ∈ H ==> X ∈ synth H"
      | Agent [intro]:  "Agent agt ∈ synth H"
      | Number [intro]:  "Number n ∈ synth H"
      | Hash [intro]:  "X ∈ synth H ==> Hash X ∈ synth H"
      | MPair [intro]:  "[|X ∈ synth H; Y ∈ synth H|] ==>
        {|X,Y|} ∈ synth H"
      | Crypt [intro]:  "[|X ∈ synth H; Key(K) ∈ H|] ==>
        Crypt K X ∈ synth H"
```

The *synth* function formalises the act of building new messages from previous knowledge. It takes as argument a message set and returns another message set. It usually takes as input a pre-processed event trace output from the other inductive relations defined so far.

The first step in the specification regards the possibility of synthesising all messages already in the knowledge set. If the knowledge set is in original form, this means a pure replay capability to the *Spy*. If the knowledge set was broken down by *parts* or *analz* it means the capability of replay and the re-sending of atomic parts. The second and third step of the specification regards the adding to the *synth* set of the guessable components for messages, which include all agent names and all guessable numbers. The fourth step of the specification regards the availability of the hash function to all message components in the *synth* set. The fifth step of the specification regards the possibility of pairing atomic message components into unlimited size, meaning that all permutations of all sizes can be yielded by the *synth* set. Finally we specify that in case keys are available, we can synthesise the encryption of all available components in the *synth* set encrypted under that key.

As we will see in Section 2.4.1.9, the *Spy* will use *synth(analz(knows Spy))* to interfere with the protocol using all his powers. Although the *synth* set contains all the possible permutations using the knowledge gathered, the admissibility of messages the *Spy* can send in a protocol is restricted by the protocol specification.

Another important property needed in protocol specifications is freshness. To capture this property we specify the function *used* as shown.

**Definition 19.** *used function definition*

```
consts
  used ::  "event list => msg set"
primrec
  used_Nil:  "used [] = (UN B. parts (initState B))"
  used_Cons:  "used (ev # evs) =
    (case ev of
      Says A B X => parts {X} ∪ used evs
      | Gets A X => used evs
      | Notes A X => parts {X} ∪ used evs)"
```

The function *used* captures all components that already appeared in our execution trace. It takes as input an event list and returns a message set. It is defined using recursion. For an empty event trace *used* returns the initial knowledge all agents have before execution. These components were used prior to the execution of our specified protocol. For the non-empty trace we take the first event and check it against the tree types of events available. In case the event is a *Says* event, we will apply the function *parts* to the payload it is carrying and unify it with the *used* set. If the event is a *Gets* event, we just return the used set, since we accounted for used components when they were sent in the first place. Finally, when the event is of type *Notes* we apply the function *parts* to the payload of the *Notes* event and unify if with the *used* set.

We are now able to specify protocols and use the above presented specifications to help us capture the subtleties of the security protocols we want to verify. Next we will show how an example protocol can be specified using the infrastructure available.

#### 2.4.1.9 A Protocol Model

Having the infrastructure's specification in place and understanding how it works, we can now start writing the specification for an example protocol using the inductive method. For that we came with a dummy example protocol as shown in Figure 2.7.

$$\boxed{\begin{array}{lcccc} 1. & A & \rightarrow & B & : \quad \{|A, B, Na|\}_{K_B} \\ 2. & B & \rightarrow & A & : \quad \{|Na, Nb, K_{AB}|\}_{K_A} \\ 3. & A & \rightarrow & B & : \quad \{|Nb\}_{K_{AB}} \end{array}}$$

Figure 2.7: Example protocol

This protocol is composed by three message exchanges or events in the inductive method terms. In the first event, agent $A$ sends a message to agent $B$ with her identity, *B's* identity and a freshly generated nonce *Na*. All this is encrypted using *B's* public key *(pubK B)* for confidentiality. At the second event $B$ replies back to $A$, with *A's* nonce *Na*, with a freshly generated nonce from himself called *Nb* and a freshly generated session key $K_{AB}$, being all encrypted under *A's* public key *(pubK A)* for confidentiality. On the third event $A$ sends it back to $B$ encrypted under *B's* public key *(pubK B)* to acknowledge the reception of event two.

The objectives of this protocol are not important for now, as we want to concentrate on its specification within the inductive method. The specification for the example protocol is shown on Definition 20.

**Definition 20.** *inductive definition of example protocol*

```
inductive_set example ::  "event list set"
where
    Nil:  "[] ∈ example"

    |Fake:"[|evsf ∈ example; X ∈ synth(analz (knows Spy evsf))|]
     ⇒ Says Spy B X # evsf ∈ example"
```

```
|EX1:  "[|evs1 ∈ example; Nonce NA ∉ used evs1|]
 ⇒Says A B (Crypt (pubK B){|Agent A, Agent B, Nonce NA|}) #
 evs1 ∈ example"

|EX2:  "[|evs2 ∈ example; Nonce NB ∉ used evs2;
   Key AB ∉ used evs1
   Says A' B (Crypt (pubK B){|Agent A, Agent B, Nonce NA|})
   ∈ set evs2|]
 ⇒Says B A (Crypt (pubK A) {| Nonce NA, Nonce NB, Key AB|})
 # evs2 ∈ example"

|EX3:  "[|evs3 ∈ example;
   Says A B (Crypt (pubK B){|Agent A, Agent B, Nonce NA|})
   ∈ set evs3;
   Says B A (Crypt (pubK A) {| Nonce NA, Nonce NB, Key AB|})
   ∈ set evs3 |]
 ⇒Says A B (Crypt (Key AB){| Nonce NB|}) # evs3 ∈ example"

|Oops:  "[|evso ∈ example;
 Says B A (Crypt (pubK A) {| Nonce NA, Nonce NB, Key AB|})
 ∈ set evso|]
 ⇒ Notes Spy {|Nonce NA, Nonce NB, Key AB|}# evso ∈ example"
```

We start the specification of a protocol by creating an inductive definition for an event list set that will model the protocol trace of events. This set will account for all the possible messages a protocol can yield and normally will be named and seen as the protocol model in the inductive method. As the protocol is represented as an inductive set, each message will be represented by an inductive step, which will be augmented by some threat model support.

The base inductive step, here called *Nil*, represents the initial empty trace. The step *Fake* brings support for the threat model. The *Fake* step involves sending $X$ generated from the application of *synth* and *analz* to *knows Spy evsf* representing the knowledge of the *Spy*. We must stress that the traces of events are constructed in reverse chronological order.

The inductive step *EX1* regards the first event of the protocol. It describes the preconditions for extending the trace with the first message. These preconditions are that the trace being extended is part of the protocol's specification and that the nonce *Na* is fresh and was not used before in the trace. If these preconditions are met, we add to the head of the trace list an event *Says* from agent $A$ to agent $B$ containing a payload encrypted under the public key of agent $B$ (*pubK* $B$). The payload consists of agent's $A$ identity, agent's $B$ identity and the fresh nonce *Na*. Note that there is no mandatory firing of any rule within the inductive description, or any order imposition other than the preconditions that are stated for the firing of any inductive step.

The next inductive step (*EX2*) regards the second event of the example protocol. Its preconditions are that the trace is part of the inductive definition for the protocol, that the nonce *NB* and the session key *KAB* are fresh and that there exists on the trace of events a message with the syntax of message one addressed to agent B. The sender can be anyone, even the *Spy*. If the preconditions are met, we extend the trace with the second event of the example protocol by adding to the head of the trace an event *Says* from agent $B$ to agent $A$ having a payload encrypted with the public encryption key of agent A (*pubK A*). We are assuming the destination of message two is agent $A$, not because it appeared as sender of message one, but because he was mentioned on its payload. This is done to corroborate the threat model since we do not trust the network by definition.

The inductive step *EX3* is the specification for event three. It is formalised with the preconditions that the trace being extended is part of the inductive definition for the protocol, that an event with syntax of message one was issued by agent $A$ before in the trace, an event with syntax of message two was issued in the trace mentioning the nonce *Na*, which agent $A$ put in message one and is addressed to him. If these preconditions are met, we extend the trace by adding to its head an event with message three syntax. Such event is a *Says* event from agent $A$ to agent $B$ containing the nonce *NB* encrypted under the session key $K_{AB}$.

Finally, the last step of the inductive definition regards the accidental loss of session keys from any principal. This is an important step in key distribution protocols to help us assert that the compromise of one session key does not affect others distributed later. The last inductive step is called *Oops* and has as preconditions that it extends the inductive definition and that message two appeared on the trace, so that a key was distributed in the run of the protocol that just happened. If the preconditions are met, the *Spy* can put on the head of the trace a *Notes* event, learning the nonce *Na* from agent $A$, the nonce *Nb* from agent $B$ and the session key $K_{AB}$ distributed in that run. Although it seems strange to hand in session keys to the attacker deliberately, this step is very important to detect the relation in compromising different session keys.

#### 2.4.1.10 Traces

Before we start looking to the properties we can verify by using the inductive method we must understand the concept of traces. Traces are event lists of arbitrary finite length that represent the execution of parallel runs for a protocol specification. They are snapshots of possible combinations our inductive definition can yield. They record the history of the underlying network while a protocol is being run. Figure 2.8 shows an admissible trace of the inductive definition for the example protocol.

```
Says Spy B (Crypt (Key AB){| Nonce NB|})
Notes Spy {|Nonce NA', Nonce NB, Key AB|}
Says B A (Crypt (pubK A) {| Nonce NA', Nonce NB', Key AB|})
Says B C (Crypt (pubK C){|Agent B, Agent C, Nonce NB|})
Says A B (Crypt (pubK B){|Agent A, Agent B, Nonce NA'|})
Says A B (Crypt (pubK B){|Agent A, Agent B, Nonce NA|})
```

Figure 2.8: Trace for Example protocol

As we mentioned earlier, traces are built in reverse chronological order. This means that older events come at the bottom while the newer ones come at the top. To help us understanding the idea of traces we go through the trace shown in Figure 2.8. Starting bottom up, we see Agent $A$ sending a message with the syntax of message one for the example protocol thus initiating a session. Then we see agent A sending a similar message again, but now creating a new session which is identified by the different nonce *Na'*. The third event in the trace shows a parallel session being started by agent $B$ with agent $C$. The fourth event shows agent $B$ answering to the second session agent $A$ requested, by sending a message with the syntax of message two of the example protocol containing the nonce *Na'*. Then we see the Spy learning the session key and nonce from B's message by the usage of an event *Notes*. The *Spy* then answers to agent $B$ with an event Says containing the syntax of message three of our example protocol.

We must remember that our inductive definition can generate traces that are of very little importance, but still admissible. We can have traces such as sending repeated times message one with different nonces without ever having a continuation in the protocol execution, or the Spy can send all the initial knowledge prior to any message being sent. These traces are admissible by our model but irrelevant for our verification towards security goals.

As traces represent protocol execution, they will be of ultimate importance in any verification effort we make with the inductive method. Sometimes the verification process can be painful, especially due to mistakes in the specification which lead to the yielding of non correct traces, as well as the lack of understanding of what is admissible as a trace for a specified protocol. Having these challenges in mind, we will try to describe in the next section how we specify and verify the protocol's security properties.

### 2.4.2 Verification of Security Protocol Goals

The ultimate aim of protocol verification is to establish whether the required security properties indeed hold for a realistic modelling and threat scenario. The inductive method allows us to specify a security protocol without limiting the number of peers, parallel runs or the size of the knowledge set for peers. But, to verify security properties in communication protocols we need first to understand them in their essence.

Goals vary from simple key-agreement properties up to the correct execution of a complex sealed-bid auction protocol. But to be able to express with precision a security goal we need first to understand the concept of goal composability.

Goal composability can be understood as the use of simpler properties as building blocks for more complex properties. We can exemplify this with a protocol that claims to achieve Authentication. To be able to assert whether or not authentication is achieved we must be able to break this goal into smaller ones.

We know that authentication can be further classified into different levels [78], but if we want to assert only the aliveness of a peer, we know that this property is not achievable by itself: we need first to assert other properties. In this example we need first to assert properties regarding the regular behaviour of the involved peers, as well as to assert the freshness of the execution. We need to lay the foundations in terms of properties it needs as building blocks. The inductive method is unique in this sense, since to be able to prove a complex goal we need to make available intermediate proofs for composing simpler properties.

The Inductive Method inherently requires a backward proof style. We specify our main goal for the protocol and then start proving the needed intermediary goals up to the point we can achieve the initial proof.

If we want to achieve a proof of Authentication (§2.4.2.7) and or Key Distribution

(§2.4.2.8), we will inherently have to prove goals for Confidentiality (§2.4.2.6), Unicity (§2.4.2.5), Regularity (§2.4.2.3) and Reliability (§2.4.2.2).

In the next subsections we will be discussing how general theorems regarding these properties can be stated and proven, as well as taking a look on the main proof strategies. We will also look into an important verification principle: Goal Availability (§2.4.2.1) which helps us to establish theorems that can be verified by the peer that needed the guarantees being yielded by the security properties.

#### 2.4.2.1 Goal availability

Before going into proving protocol properties, we must first touch on how to state them. Proving properties regarding security protocols is complex and labour intensive using the inductive method. To avoid spending our resources into proving properties that are either not meaningful or useful we must follow some goal specification principles.

Starting by Bella's summary of the Goal availability principle [20] he says, "Goal availability tells us that formal guarantees must be studied from the agents' viewpoints to check if the peers can apply them within their minimal trust". So the Goal Availability principle is our way to address whether a security property being stated is meaningful and is available for the peer needing such a guarantee. To be able to fully understand the principle of Goal Availability we need to define some of its terms.

Minimal trust is the minimal set of facts available in the formalised environment for a protocol, whose truth values an agent needed to know in order to derive the guarantees, but which he cannot verify in practice. In lay terms, minimal trust is the set of knowledge we cannot derive, but we have to believe.

Other important concepts for understanding Goal Availability are the ideas of an available goal and an applicable guarantee. An available goal is a goal where there exists an applicable guarantee in the protocol specification that confirms the goal to the peer under the protocol specification. An applicable guarantee is established on the basis of a guarantee an agent can verify within his own minimal trust.

The goal availability principle gives us the guidance on how to specify the properties we want to verify. Furthermore, it enables us to have a thorough approach in how to address the statement of such properties in a way that will enable us to find more subtle problems while guaranteeing their meaningfulness. Although we recognise the importance of this principle, some of the examples below lack that. Some are intrinsically complicated to specify in term of the agents' point of view, while others are simply more educational than their Goal Availability ready counterparts. We also must understand that some properties are just environmental and can only be seen by "God's eye".

#### 2.4.2.2 Reliability

Reliability lemmas are not proven to directly achieve any security property for any protocol, but to assert how reliable the modelling is regarding the system it represents. The objective of such lemmas is to assert known systematic construction properties that must hold for the protocol we are working on. Furthermore, they may represent system wide rules that we can derive in order to help us with the verification process. These system wide derivations are available for us to help with proving more complex protocol goals.

Some reliability lemmas are proven within the method, which makes them available in all protocols we verify. They represent system wide rules that we can assert regarding the inductive method infrastructure. There are numerous lemmas stating such reliability properties already available in the inductive method implementation. We can technically state that most of the theorems available in the theories Message, Events and Public, which are the core of the inductive method main strain, are indeed reliability lemmas. To elucidate their general form we will discuss some.

The operators of the inductive method, namely *parts*, *analz* and *synth* have some useful properties that once proved help us to establish their usability in terms of real representation they are supposed to represent. We can show a rule for the *parts* operator that tells us that the atomic parts in the trace before its extension is a subset of the atomic parts after the extension. This fact means that no parts of a message are left behind when a trace is extended.

As stated before the number of reliability lemmas generically available within the method is big. But not everything in terms of reliability can be generalised. Some reliability lemmas are proven regarding protocol features that we want to show we captured in our specification effort or that represent structural protocol features.

A possibility lemma states that we can achieve the execution of the last message in our protocol. The usefulness of such lemma is twofold. First it helps us to capture specification problems, such as preconditions that we did not correctly stated or that make the firing of some rule unachievable. The second use for such lemma is the achievement of a weak aliveness, since we can state that there is at least one trace that reaches the end of the protocol. Lemma 1 show the Possibility lemma for the Needham-Schroeder Shared Key Protocol, which we will use mostly throughout this section due to the the fact it is a well known example for people studying security protocols.

**Lemma 1.** *Possibility NSSK*

```
[| A ≠ Server; Key K ∉ used []; K ∈ symKeys |] ==>
   ∃N. ∃evs ∈ ns_shared.
      Says A B (Crypt K {|Nonce N, Nonce N|}) ∈ set evs
```

Proving such a possibility lemma is straightforward. We just need to join the protocol rules in our inductive specification so that all their preconditions can be met. The proof in this case involves some preparation steps and the application of the simplifier from Isabelle/HOL.

Another form of reliability property regards stating that agents other than the Spy follow the protocol rules. In reality all agents want to achieve their protocol goals. This implies that if they are not the *Spy* or they are not colluding with the *Spy* they behave following the rules the protocol established. One usual type of reliability proof for agents in a shared key session key distribution protocol regards the establishment of the reliability of the trusted third part. Lemma 2 shows us the reliability of the Server in the Needham-Schroeder Shared Key Protocol.

**Lemma 2.** *Says_Server_message_form*

```
"[|Says Server A (Crypt K' {|N, Agent B, Key K, X|}) ∈ set evs;
   evs ∈ ns_shared|] ==>
      K ∉ range shrK ∧
      X = (Crypt (shrK B) {|Key K, Agent A|}) ∧ K' = shrK A
```

Lemma 2 states that if a message sent from the server with syntax of message two is in the trace of events, and the trace is generated by our inductive specification, then the session key $K$ is not on the range of long-term shared keys, and the certificate $X$ has the correct form by mentioning key $K$ and agent $A$ and being encrypted to be forwarded to agent $B$, and finally that the server encrypted the whole message with the key of the intended recipient, in this case $A$.

This lemma tells us that the *Server*, who by definition is not in the set *bad*, creates a coherent protocol message following the agreed specification. Although this lemma states the reliability of the server regarding the message composition, it does not state the reliability of the property that sessions keys are fresh [20]. For that we need a new lemma that states the freshness of sessions key based on the idea of event ordering introduced by Bella with the function *before* and the usage of the function *used.* We will not cover this reliability property in this thesis.

The variations of reliability properties that can be stated for specific protocols are vast and cannot be enumerated exhaustively. Bella in his extensive research with the method [20] produced some interesting examples regarding reliability properties, especially in the smart-card and accountabilities scenarios.

The principle behind these properties is clear since we want them to establish rules that show how reliable is our protocol specification regarding the real protocol. Such properties tend also to be stated and proven so that we can reuse them in other more complex properties such as Confidentiality and Authentication. So a rule of thumb for the minimum properties that should be proven in this category is that we should state and prove at least the properties we need to conclude the security goals of the protocol being verified. This is normally achieved by the backwards proof strategy inherent to the inductive method.

#### 2.4.2.3 Regularity

Regularity properties are properties we specify regarding the rules for messages or message components that appear in the trace of events. In general regularity lemmas are the bases over where we state our authenticity and authentication properties. A simpler way of stating what regularity lemmas are is stating that they represent any property that can be asserted for a message by the fact that it appeared in the traffic. As an example we can cite Lemma 3 for Lemma 2 above.

**Lemma 3.** *NS3_msg_in_parts_spies*

```
Says S A (Crypt KA {|N, B, K, X|}) ∈ set evs ==>
   X ∈ parts (knows Spy evs)
```

Lemma 3 states that in a message from an Agent $S$ to an Agent $A$ with the form of message two in the Needham-Schroeder Shared Key protocol, the certificate $X$ is in the knowledge of the *Spy.* This fact has no novelty in itself, since by definition of *parts* we know that it is true. This regularity lemma is very useful when we are trying to prove properties regarding the certificate itself, since it transforms the event *Says* into the existence of the encrypted certificate in the knowledge of the *Spy.*

Regularity properties are also very important to help us prove that messages can be considered tamper-proof for authenticity and confidentiality properties. A specific type of regularity property is key regularity, which states that if a long-term key, either shared or private, is sent on the traffic this implies the agent owning that key is not behaving in accordance with the protocol. An example of such key regularity property can be seen on Lemma 4.

**Lemma 4.** *Spy_analz_shrK*

```
evs ∈ ns_shared ==>
   (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)
```

Lemma 4 is the key regularity property for the Needham-Schroeder Shared Key protocol. Its proof is based on the fact that the only way for a long-term key to be sent on traffic would by the rule *Fake*. In the inductive specification for the protocol, that means that only the *Spy* would have done that. The importance of this lemma appears when we want to prove communications are tamperproof by the usage of *A*'s secret key, since that we can reduce the preconditions to *A* colluding with the *Spy* instead of looking for his key on traffic.

The importance of this lemma according to Bella [20] lies on the fact it translates a condition an agent cannot verify, such as his key long-term key being sent on the traffic, to one he is able to verify which is his collusion with the *Spy*. This is of importance for the goal availability principle as we saw before on section 2.4.2.1.

But the existence of key regularity properties in protocols is paramount since without such guaranty we cannot establish most of the security goals for any given protocol.

#### 2.4.2.4 Authenticity

Authenticity for a message is coupled with integrity. In general to consider a message authentic means to confirm its originator, but authenticity is not a property that lives on its own, as stated by Anderson [6]. Authenticity is entangled with integrity by the fact that a message cannot be claimed authentic if it was modified in transit. Also to establish whether a message has integrity we need to establish its last authorised modification, what inherently leads back to the authorisation and to the confirmation of its originator.

Due to this intrinsic relation, in the inductive method, to assert a message has authenticity means to assert it has integrity. It is also established that to confirm the authenticity of a message we can establish it is tamperproof by the use of encryption. Authenticity will hold if the keys used for encryption are not compromised in any form.

To state the authenticity of the certificate *B* receives in message three of Neeedham-Schroeder Shared Key protocol, we need to assert it in terms of the integrity provided by the encryption with *shrK B*, which is the key shared between *B* and the *Server*.

**Lemma 5.** *B_trusts_NS3*

```
[|Crypt (shrK B) {|Key K, Agent A|} ∈ parts (knows Spy evs);
  B ∉ bad; evs ∈ ns_shared|]
    ==> ∃NA. Says Server A (Crypt (shrK A) {|
            NA, Agent B, Key K, Crypt (shrK B) {|
                Key K, Agent A|}|}) ∈ set evs"
```

Lemma 5 only holds because of the assumption $B \notin bad$, since this guarantees the integrity of the certificate by not giving the *Spy* access to the shared key between $B$ and the *Server*. The proof of this lemma is straightforward since we will just prepare and apply induction followed by referring to regularity lemmas 3 and 4.

Authenticity properties are affected by the principle of goal availability (§2.4.2.1). To prove authenticity and integrity for a message over assumptions an agent cannot verify is not useful. Sometimes domain specific issues can complicate proofs. We can have the assertion of authenticity for a message encrypted under session keys, since their authenticity is related to the authenticity of how they were distributed in the first place. Another example is the authenticity of pair keys in the verification of Shoup-Rubin [20], where the assumption of secure means affects the construction of the authenticity properties.

#### 2.4.2.5 Unicity

Unicity properties are related to the freshness of components used in messages. Freshness is normally asserted over nonces, timestamps and session keys so that we are able to know that we are not being fooled into a replay and that no other runs of the protocol would be able to receive the same session keys as we did.

In the inductive method, we represent freshness by asserting the uniqueness of such message components within a context. From uniqueness we consider the binding of such fresh components only to the singular context we first used and nowhere else. We have two ways of expressing freshness by unicity in the inductive method. The original one introduced by Paulson is demonstrated by Lemma 6.

**Lemma 6.** *unique_session_keys*

```
[|Says Server A (Crypt (shrK A)
        {|NA, Agent B, Key K, X|}) ∈ set evs;
  Says Server A' (Crypt (shrK A')
        {|NA', Agent B', Key K, X'|}) ∈ set evs;
  evs ∈ ns_shared|] ==>
          A=A' ∧ NA=NA' ∧ B=B' ∧ X = X'
```

The original way of asserting the freshness of a nonce or a session key in the inductive method was the establishment of a property like Lemma 6 for the required component. Lemma 6 shows us the asserting of freshness for the session key generated by the trusted third party in the Needham-Schroeder Shared Key protocol. This lemma is important because it asserts the total reliability of the Server: he composes message two correctly by always using fresh session keys. Lemma 6 states that if there are two events with the syntax of message two where a key $K$ appears, all other components must be the same. The proof of Lemma 6 is simple. We prepare and apply the induction followed by the Isabelle/HOL simplifier and classical reasoner. We should note that we do not need to assert anything regarding the compromising of the agent *Server* because it is trusted by definition.

Although the previous lemma is able to assert freshness of a session key by locking it to a unique context, Bella noticed during his experimentations [20] that it allows the multiple sending of message two in this case. This would not necessarily affect the security of a protocol based on nonces, since this would account only by a retransmission based in a correction from the transport layer for example. Although the usage of such type of lemma in a protocol based on timestamps would lead to some inaccuracies regarding the representation, since uncompromised agents would always send the correct timestamp. To filter out this inaccuracy Bella proposed the creation of the predicate *Unique*.

**Definition 21.** *definition of predicate Unique*

```
definition
  Unique ::  "[event, event list] => bool" ("Unique _ on _")
    where "Unique ev on evs =
          (ev ∉ set (tl (dropWhile (%z.  z ≠ ev) evs)))"
```

The predicate *Unique* (Definition 21) is a predicate taking an event and an event list returning true in case the event does not appear more that once in the event list. It is implemented using some list processing functions. The *dropWhile* function will scan a list until an element is found, while function *tl* will prune into shape the result for the set membership application with the event we want to determine its uniqueness. An example of the usage of the predicate *Unique* is shown in Lemma 7 where we rewrite the uniqueness Lemma 6 to use the predicate.

**Lemma 7.** *Unique_session_keys*

```
evs ∈ ns_shared ==>
  Unique ( Says Server A (Crypt (shrK A)
                {|NA, Agent B, Key K, X|}) ) ∈ set evs
```

### 2.4.2.6 Confidentiality

Confidentiality is defined [51] as the non disclosure of secret components to agents not intended to receive it. In the inductive method, confidentially is the asserted by the not appearance of certain terms in the knowledge set for the Spy. This should not happen after cryptographic analysis under the terms the *Spy* is allowed to operate by the threat model. In other terms, we will consider the confidentiality argument being stated as $X \notin$ *(analz( knows Spy evs))*.

Confidentiality is one of the usual security properties we find in security protocols. It will be the cryptographic basis we will build our other properties on top of. For example, to be able to assert authentication, we generally need first to establish the confidentiality argument for the components providing us with such authentication. To exemplify a confidentiality property, we show the secrecy argument for the session key being distributed in the Needham-Schroeder Shared Key protocol.

**Lemma 8.** *secrecy_lemma*

```
[|Says Server A (Crypt (shrK A) {|NA, Agent B, Key K,
  Crypt (shrK B) {|Key K, Agent A|}|}) ∈ set evs;
  A ∉ bad; B ∉ bad; evs ∈ ns_shared|] ==>
   (∀NB. Notes Spy {|NA, NB, Key K|} ∉ set evs) -->
     Key K ∉ analz (know Spy evs)"
```

Lemma 8 states that if evs is an admissible trace of the Needham-Schroeder shared key protocol specification, and an event with the syntax of message two appears on the trace for a key $K$ where the agents $A$ and $B$ mentioned in such message are nor colluding with the *Spy*, this implies that if the *Spy* did not learn the key by a leak with an *Oops* event, the key $K$ is not available to him.

Proving Lemma 8 requires the usual inductive strategies. They are followed by the regularity arguments regarding message two and a simplification augmented with facts regarding the *analz* operator idempotence properties. We are left with the sub goals generated by the inductive steps from message three (*NS3*) and the *Oops* event. For both we must use the argument for unicity of session keys (Lemma 6), and for the sub goal from *NS3* we must further refer to the authenticity argument for message NS2[1] and the injectiveness of the operator *analz* so that they can be eliminated and the confidentiality argument for session keys be achieved.

Conducting proofs regarding confidentiality properties is generally one of the most difficult tasks we have when verifying a protocol. This happens due to the way the *analz* operator is constructed. Its construction, as shown in Definition 17, helps us with symbolic evaluation in all components of the list of events and rules them out if they do not include any key material. When including key material, we are left with sub goals that require us to prove the uncompromising of each key stated or the uncompromising of its counterpart. Some of these key compromising lemmas are straightforward, while some others are not. We will see that in our experimentation with non-Unicast events, this issue with the construction of *analz* had to be addressed in a rather different way to enable us to do symbolic evaluation of the terms and achieve proofs under the new event model.

#### 2.4.2.7 Authentication

Agent authentication is defined by ISO/IEC 9798-1 [72] as "mechanisms (that) allow the verification, of an entitys claimed identity, by another entity. The authenticity of the entity can be ascertained only for the instance of the authentication exchange". Agent authentication is one of the main goals of security protocols and among the most important theorems expressed with the inductive method.

But to thoroughly describe authentication we must first refer to Lowe's hierarchy of authentication specification [78]. Lowe defines four increasing levels of agent authentication. The levels suppose a protocol initiator $A$ completing a session of the security protocol with the responder $B$. If $A$ is able to assert the *aliveness of B* this means that she believes $B$ has participated in the protocol. If $A$ believes *the weak agreement of B with herself* this means she believes $B$ has been running the protocol with her. *Non-injective agreement of B with A on H* means a weak agreement between $A$ and $B$ agreeing on the set of message components *H. Injective agreement of B with A on H* means a non-injective agreement that happened only once. The inductive method is able to verify and distinguish between each one of these classes of authentication. The extension proposed by Bella [20], such as message reception and the principle of Goal Availability, played a very important role into establishing this.

Authentication in the inductive method can be interpreted as determining the true

[1]Not shown in this thesis

originator of the exchanged authentication token. One important definition for us to assert the true creator of a message component $X$ is the predicate *Issues*, which returns true if an agent $A$ issues another agent $B$ with $X$ and $X$ never appeared on the trace prior to that event.

**Definition 22.** *definition of predicate Issues*

**definition**
```
   Issues ::  "[agent, agent, msg, event list] => bool"
      ("_ Issues _ with _ on _") where
            "A Issues B with X on evs =
      (∃Y. Says A B Y ∈ set evs & X ∈ parts {Y} &
      X ∉ parts (knows Spy (
            takeWhile (% z.  z ≠ Says A B Y) (rev evs))))"
```

Definition 22 creates a predicate that takes two agents, a message and an event list, and returns a boolean representing if A is the true creator of the message and it was intended to the agent $B$, which is the receiving counterpart. The predicate is implemented using a syntax annotation to make it more readable and states that if there exists a $Y$ that is the payload of an event Says originating from agent $A$ to the destination agent $B$ to which $X$ belongs to its set of *parts*. The predicate will be true if this first argument is true and also if $X$ is not in the parts of the reversed trace under another *Says* event.

Knowing the true creator for a message component is the way the inductive method uses to assert that some agent can authenticate another agent by the uniqueness of the message he/she issued. To exemplify the usage of the predicate *Issues* we will look to Lemma 9 which will assert the premises for us to authenticate $B$ as the true creator for message four in the Needham-Schroeder Shared Key protocol

**Lemma 9.** *B_Issues_A*

```
[| Says B A (Crypt K (Nonce Nb)) ∈ set evs;
      Key K ∉ analz (knows Spy evs);
      A ∉ bad; B ∉ bad; evs ∈ ns_shared |]
            ==> B Issues A with (Crypt K (Nonce Nb)) on evs
```

Lemma 9 states that agent $B$ authenticates himself to agent $A$ using the session key to encrypt his own nonce in message four, $B$ has some preconditions. First there should exist in the trace of events a message with the syntax of message four from agent $B$ to agent $A$ with the payload being the nonce *Nb* encrypted with the session key $K$. Then the key $K$ should not be compromised being in the knowledge of the *Spy*, nor agents $A$ or $B$ could be colluding with the *Spy*.

To prove this lemma in Isabelle/HOL we first need to expand the definition of the predicate *Issues*. Then we eliminate the quantifiers and prepare and apply the induction over the protocol specification. The proof follows by preparing the treatment of the cases $X \notin$ *analz (knows Spy evs)* and applying simplification. We are left with three trivial sub goals: one for the *Fake* case, one for message three and one for message four. The *Fake* case is solved by the classical reasoner itself. Message three needs some information regarding the function *takeWhile* and explicit appeal to regularity and authenticity lemmas regarding the *Server* behaviour and message two respectively. Sub goal for message four is sorted out by the classical reasoner augmented with information regarding the *takeWhile* function.

As shown in Lemma 9, the Inductive method is capable of verifying non-injective agreement of $B$ with $A$ on *Crypt K (Nonce Nb)*, which is one of the major goals of the Needham-Schroeder Shared Key protocol. In this case, due to the modelling adopted and the intrinsic characteristic of the protocol using nonces we cannot establish a stronger class of authentication.

#### 2.4.2.8 Key Distribution

Key distribution concerns the establishment that, whether, at the end of a protocol session, the peers have sufficient evidence that they share a session key with the intended peer. This is normally a major goal for a security protocol. Such evidence of agreement in a session key is embedded by protocol designers using the distributed session key in a challenge response. By doing that, each side infers that the other has knowledge of the key distributed. Lemma 10 states such evidence for key distribution from agent $A$ to agent $B$ in the Needham-Schroeder Shared Key protocol.

**Lemma 10.** *A_authenticates_and_keydist_to_B*

```
[|Crypt K (Nonce NB) ∈ parts (knows Spy evs);
Crypt (shrK A) {|NA, Agent B, Key K, X|} ∈ parts (knows Spy evs);
  Key K ∉ analz(knows Spy evs);
  A ∉ bad; B ∉ bad; evs ∈ ns_shared|]
          ==> B Issues A with (Crypt K (Nonce NB)) on evs"
```

This lemma strengthens the authentication lemma shown before (9). It also uses the predicate *Issues* for determining the true creator of the payload for message four. As this lemma is based on evidence agent $A$ has regarding the execution of the protocol, it states as preconditions that the payload of message four is in the trace of events and the payload of message two is in the trace of events as well. Then the session key $K$ should not be known by the *Spy* by the use of his cryptographic powers over the trace of events, as well as agents $A$ and $B$ not being colluding with the *Spy*. If these preconditions are met we can assert that $B$ is the true creator of the encryption of his nonce *Nb* with the session $K$ confirming the evidence that he has the key that was distributed during the protocol run. This is guaranteed by the assumption from message two. Proving this lemmas is straightforward since we can appeal to Lemma 9 for most of its resolution.

This lemma could be strengthened even further if we adopted the message reception primitive introduced by Bella [20]. Instead of inspecting the trace with the predicate *Issues* and the preconditions being asserted with trace inspection, we could assert that message two and message four we received and that *Crypt K (Nonce NB)* is in the knowledge set of agent $A$, a precondition easier for him to verify than the ones stated above.

## 2.5 Considerations

In this chapter, we reviewed the field of security communication protocols, focusing on their design and verification areas. We started by covering the basic principles behind protocol design. We covered the cryptographic basis and the way security protocols are represented. We then overviewed the attacks these protocols normally suffer, focusing on how their goals are defined and to which threat model they are normally subject to. Some attacks on classical security communication protocols were studied.

On the protocol verification side, we covered why formalisation is the best way to achieve a thorough review needed to confirm any claim made regarding the represented protocols. We also introduced the various different approaches developed so far. These approaches are focused on the Unicast message casting primitive and are divided in two main strains: *State Exploration* and *Theorem Proving*. *State Exploration* techniques allow us to easily find bugs in protocols, while *Theorem Proving* techniques help us assert the existence of security properties in protocols. We also tried to cover some details of the most important tools available for each approach.

We then focused on Paulson's inductive method which is the method of choice for this thesis. In the Inductive method overview we covered the construction of the basic infrastructure it needs. We also covered how to specify the protocols for verification. We briefly introduced the ideas behind traces of execution and the principle of Goal Availability. We then characterised once more protocol goals and went through examples of verification done with the inductive method and Isabelle/HOL for each one of the examples.

With these concepts in mind and making the clear distinction that most of these methods are not prepared for what we propose, we will bring in the next chapter our first main contribution for this thesis. This contribution is the modelling of a new events theory that is able to verify other message casting methods than Unicast. The novelty is clear since none of the available tools today claim the verification of security protocols under these different network casting frameworks.

— *We have two ears and one mouth so that we can listen twice as much as we speak.*

Epictetus (AD 55-c.135)

# 3 Security Multicast Protocol Verification

Multicast was initially advertised as a scheme for better network resources usage [105] and for maximising the user experience when receiving content that could be easily replicated. Although Multicast infrastructure is today a reality its usage and application remains stigmatised by these initial assumptions. Multicast had this first conception but we have seen increasing interest in it from the security community. Initially with protocols for secure content delivery [37, 103] trying to address specific multicast problem and later on in protocols that involve Byzantine Agreement and Byzantine Security [76] taking the advantages of the new message casting framework.

In the last two decades multicast evolved to fulfil more that its initial performance-based aim. It is now being developed much more towards reliability. By increasing the reliability of multicast based protocols, the number of applications that can benefit from it is also increased. As examples we can cite the usage of multicast based communication in the stock trading business. The versatility of the Multicast implementation is appraised by the fact it is the basic building block we can use to construct all other message frameworks known. Once the complexity of achieving properties such as reliability can be addressed for a multicast framework, its application for other message casting frameworks is generally straightforward.

In the security protocols side, we have seen new strains of protocols based on unicast, multicast, broadcast and a mixture of the three modes. We can cite numerous examples, such as protocols to assure secrecy on one-to-many communications [65], protocols to guarantee authenticity in one-to-many communications [60, 125], key distribution in one-to-many communications [66, 32], and protocols that deal with novel security goals such as byzantine agreement [47, 69, 12, 127], multi-party computation [33] and digital-rights management [37, 103].

The verification process for such protocols must match the development done by designers. Some efforts were seen in the literature, but they generally have problems to cope with the inherent complexity of one-to-many communication. One-to-many message casting models inherently increase the size and complexity of the knowledge sets of peers as well as the size of the representation of the execution, which are two of the limitations explained in chapter 2. We have seen some efforts being made using model checking using CSS based approaches [5, 64] as well as some using theorem proving, in particular using the NPA approach [7] and using Graham Steel's Coral [118].

The implementation aspects of what Steel did with Coral in the verification of multicast based security protocols confirm the great potential of the inductive method in addressing the problem. Our aim, then, is to extend it to enable reasoning regarding the multicast-based event primitive. Our main goals are to create a versatile event model that can encompass multicast and all the other variations of message casting that multicast can yield.

Our idea in this chapter is to do a brief review about general multicast protocols (§3.1), and show the reader why it is a good strategy for augmenting the coverage of the inductive method to have a Multicast-capable event theory. We will discuss the different types of multicast frameworks security protocols normally require (§3.1.1, 3.1.2 and 3.1.3). Then we will see our contributions in the extension of the inductive method towards having a fully capable multicast event theory (§3.2). We will also introduce the basic lemmas required to make the implementation usable. Then we will go over a verification of a well known security protocol under our newly designed framework (§3.3), so that we can corroborate its usage in different message casting scenarios. We also revisit this protocol so that we can measure if the proposition of adapting the inductive method towards a Multicast based event theory is feasible in terms of the effort introduced in the verification process. We will conclude with an analysis of the new verification capabilities the inductive method has after these extensions.

## 3.1 General Multicast Protocols

Multicast is a one-to-many message casting framework that was originally designed to selectively deliver messages to specific nodes in a network while maximising efficiency, avoiding replication of traffic. Multicast aims to use the network layer in a very efficient way by requiring the source of the cast to issue the packet only once even if it needs to be delivered to a large agent population. The network is in charge of doing the necessary replication and enabling the delivery of the payload to all the agents within the multicast group. The network layer is responsible for providing the infrastructure necessary for the multicast to happen.

A multicast group is established by the creation of a multicast session, where an initiator creates a session by obtaining a name space to it. In an IP network, it is a specially reserved address, while in other networks, it is only a reserved name in the namespace. After naming the session, the initiator will define its characteristics, which can be the multicast type, scope and duration. With the announcement of the session, peers may join or be added to the session. The network infrastructure needs to be informed about the multicast group shape and form. With the session in place, we can start to multicast to the group defined by the session we just created, knowing that our message will be sent to the intended recipients in a very efficient way. Peers are allowed to leave the session at any time, giving to the sessions a dynamic shape.

Normally, due to application constraints multicast is implemented in an unreliable way. The usual application of transmitting data streams for content delivery [75] makes efficiency the only objective of multicast session. This creates the requirement for a very efficient transport layer which, to achieve that, cannot afford to create connection controls for delivering packets reliably. This lack of reliability is defined by the loss, reordering and multiple delivery of packets to the endpoints. The application layer is responsible for dealing with these subtleties by reconstructing, reordering and dropping eventual data stream problems.

Other applications require multicast to behave as a network layer where lossless transmission, non-duplication of content and ordering need to be enforced. Security protocols are generally among these applications. To be able to address that, we see the inception of a classification for multicast, dividing them in three categories. The first class of multicast is the above explained unreliable multicast, where messages can be lost, permuted and multiply delivered. Some security protocols can cope with these properties.

The second class of multicast is called Reliable Multicast. Reliable Multicast schemes, which are the basis for Byzantine Agreement [76], ensures that to all honest members of a given multicast group in the network will be delivered the same message, even in the face of a hostile presence in the network layer and among the group of multicast initiators. The loss of messages is prevented. Furthermore, it introduces the usage of novel transport layers so that this guarantee can be achieved in the one-to-many scenario.

Atomic Multicast schemes are an extension to Reliable Multicast that enable honest members to have messages delivered in the order that they were sent. Moreover, Atomic Multicast schemes guarantee the delivery of messages only once. This category of multicast is very difficult to achieve and normally yields a security protocol in itself, since it requires the existence of some basic security properties, such as uniqueness and reliability.

In the next subsections we will be looking over the important characteristics of the three classes of multicasts. We will try to capture their properties for a modelling of a multicast framework in the inductive method. This will help us to extend the coverage of the inductive method to novel protocols and novel security goals, paving the way for the verification of Byzantine Agreement protocols based on Reliable Multicast. Subsection 3.1.1 will give us an overview of the properties for Unreliable Multicast and how should we achieve that in the inductive method. Subsection 3.1.2 will enlist the properties of Reliable multicast schemes and Subsection 3.1.3 will do the same for Atomic Multicast.

### 3.1.1 Unreliable Multicast

Unreliable multicast is the classical multicast framework for IP Multicast [105, 4] designed to use the network infrastructure in an efficient way. The implementation of such Multicast network base layer is based on the usage of very basic transport layers. The idea of the unreliable multicast is the provision of the multicast naming framework, where sessions can be established and where addresses for representing the various sessions can have the translation from this meta destination into the peers that belong to the group.

An important characteristic of Unreliable multicast that will extend to the other classes is that its operation does not require the issuer of the multicast communication to understand or know the addresses of the destination peers, since that is taken care of by the network infrastructure. At the application level this knowledge can be a requirement. In fact the purpose of unreliable multicast is solely to provide a way so that the sender can send a single message that will be replicated when needed towards the destination nodes. Our multicast representation will use a naming space for the multicast groups being represented by an unlimited size list. This representation will enable us to abstract to the sending peer the actual recipient peers. When necessary we can also make specific mention to peers in the various concurrent sessions happening.

Unreliable multicast in the form of IP multicast is widely deployed in the Internet with applications ranging from video-conferencing and video content distribution [103, 37] to examples of low latency needed in business such as stock exchanges [84].

With a multicast infrastructure, we can capture the delivery of communication with all the message casting 99frameworks known today in computer networks. A multicast to a multicast group of size one is a proper representation for a Unicast. A multicast to a group of size all is also a proper representation of a Broadcast. Multicast is so versatile that istcan encompass even the very new message casts frameworks' variations, such as Anycast and Geocast. Anycast is a message delivery system that delivers contents to be readable by a single peer among the ones belonging to the multicast group, it is a one-to-one-of-many association [3]. Geocast is a specialised multicast that takes into consideration the geographical availability of the receiving peers in a Multicast Group [71]. In fact most security protocols based on Multicast require in fact the representation of Anycast to be achievable.

Our aim in this thesis is to produce a new events theory model for the inductive method that is capable of representing by multicast all the other network casting frameworks. As we will show later in Section 3.2, this is achievable by embedding the representation of the group within the message casting while enabling the capabilities of Anycasting by the usage of a $\lambda$ operator over the message payload. This is coupled with the above idea of representing the multicast group as an unbounded list. To prove our concept is capable of cross framework representation, we revisit the Needham-Schroeder Shared Key protocol implemented by a message framework composed by a Multicast to a group of size one in section 3.3.

### 3.1.2 Reliable Multicast

The aim of a reliable multicast scheme is to ensure that group members of a multicast group receive the same message that was sent with no loss of packets. Ordering is not part of the concept but is achieved by many implementations.

One of the main issues regarding the classification of reliable multicast schemes is the definition of reliability. Reliability itself in this context means that to every packet sent a packet will be received by an endpoint participating in the multicast session. But the implementation of reliability is not so straightforward [56] since the implementation of multicast does not require the sender to know each of the destination peers address and does not enable the sender to authenticate any recipients.

Reliable Multicast can be achieved in the inductive method by the usage of a message reception case in the inductive specification of a protocol, so that we can confirm the reception of the the message by all the peers involved in the multicast session. We modelled that in the inductive specification of the Franklin-Reiter Sealed Bid auction protocols, but we did not use it in our verification, as we will show later in Chapter 5. We opted not to create a specific reception constructor in the inductive method since we can achieve a reliable multicast scheme by enforcing all the multicast session participants get the information they need using the standard *Gets* constructor. This option was made to avoid introducing new complexity to the verification.

### 3.1.3 Atomic Multicast

The aim of atomic multicast is to extend reliable multicast so that we can guarantee the unicity and ordered delivery of messages to the multicast group [49]: delivery is guaranteed and it is done only once.

Atomic multicast methods are important in theory but not in practice [49]. We have a series of methods developed especially for atomic broadcast, which are a special case for multicast, but very few of them are available in practice due to the non availability of the preconditions needed to execute such protocols. This hinders the development of novel protocols.

Having pointed out the characteristics for Multicast communication in the previous sections, we will see next our implementation of a multicast support for the inductive method. We will show our extensions and the inclusion of support facts to our implementation so that we can assure the coverage of our multicast implementation as suggested in section 3.1.1. We will then show an example that corroborates that our implementation is capable of representing all message casting frameworks known today by applying that to a well known protocol (§3.3) to measure if it is feasible to change the message representation framework for the inductive method so that more protocols can be specified.

## 3.2 Inductive Method extensions for Security Multicast Protocols

The inductive method initially allowed only three formal events, which formalise the act of sending, noting and receiving a message. The original implementation of the event datatype can be seen in Definition 23.

**Definition 23.** *Original event datatype definition*

```
datatype
  event = Says agent agent msg
         |Gets agent msg
         |Notes agent msg
```

The original definition for the event datatype (23) allow us only the formalisation of three types of events. The first event formalises the act of an agent sending a message to another agent through the use of the primitive *Says*. The second event, introduced by Bella's Goal availability and peers knowledge ideas [20] formalises the act of receiving a message by an agent through the usage of the primitive *Gets*. And finally, the third event formalises the act of an agent receiving information off-protocol, where he just writes down this knowledge for future use through the usage of the primitive *Notes*.

The idea of Multicast communication explained in the previous section (3.1) allowed us some choices in how to implement a Multicast event. In our first implementation attempt, we tried implementing the Multicast communication as a series of Unicast communications from the sender to all recipients in the multicast group. Although this represented clearly an extension to the model without interfering with the datatype structure in place, it introduces a series of infidelities to the real implementation of Multicast and makes reasoning harder when identifying a specific Multicast message is a necessity.

Infidelities to the Multicast real implementation generally regarded the impossibility of implementing different types of Multicasts as explained in Section 3.1, and the ability of distinguishing a Multicast from a Unicast after the translation has happened. We also had issues of fidelity with the idea of Multicast, where a message is sent once and is received multiple times with a main goal of reducing network traffic or to enable the replication of data. If implemented in this way, a Multicast would mean the sending of multiple messages, what would make it different to the real implementation yielding undesired properties.

Therefore we decided to implement an extended datatype event, creating a new primitive Multicast that is able to represent a Multicast communication with fidelity and also to encompass the possibility of implementing the cases for Unicast and Broadcast and the other casting frameworks using the Multicast primitive.

**Definition 24.** *Multicast event datatype definition*

**datatype**
```
  event = Says agent agens msg
         |Multicast agent "(agent list)" "( agent => msg)"
         |Gets agent msg
         |Notes agent msg
```

The new datatype of events introduced in Definition 24 implements the same primitives as the original event datatype shown on Definition 23. This is done for the sake of compatibility with the actual Unicast protocol verification, meaning our theory can be directly exchanged by the actual Events theory available with Isabelle/HOL distribution. If you do not invoke the usage of any Multicast message, the verification should remain mostly unchanged.

We add to the datatype event a new primitive *Multicast*, where an agent multicasts a message to a multicast group. Our technical choice for representing the multicast group by using an agent list is due to the fact that lists are a powerful representation in Isabelle/HOL and come with an extensive implementation. The idea of representing the message as a function over agents has direct inspiration from the real Multicast communication and the necessity of implementing Anycast, as most security protocols requires that. In a real scenario, a peer sends the same message to a group of receivers, and each receiver is capable of interpreting the message in different ways, sometimes depending on the knowledge he/she already has.

Another source of inspiration for letting each peer apply a function with his own parameters is that this creates their own view of the message. By applying the function to his own identity the agent is able to view the content directed to him. This implementation follows the core idea of Goal Availability [20], since we are working on extending Agents' knowledge independently. The implementation details will make the issue clear in the next subsections (3.2.1 and 3.2.2) where we extend the knowledge of peers based on their point of view inside or outside of the multicast group and the extension of the set of used components allowing us to reason about the trace with all points of view added to it.

Although it seems contradictory to reject an implementation based on Unicast to represent Multicast and implement a Multicast datatype capable of representing Unicast and Broadcast, this implementation does not suffer the setbacks of the previous ideas. We also wanted to create a generic implementation to corroborate our motivational idea that Unicast and Broadcast are extremes for Multicast, setting Multicast as the base implementation for verifying Security Protocols in the future.

In the next subsections we will show the extensions made in how peers acquire knowledge (§3.2.1) and how we can have access to all derivations coming from the application of the function to each peer's point of view through the function *used* (§3.2.2). Following on we will discuss the modification in previously available lemmas and the new lemmas created to support the usage of our implementation (§3.2.3).

### 3.2.1 Extending Peers' Knowledge set for Multicast Communications

Peer knowledge and the implementation of the set *knows* in the inductive method were important additions to enable us to reason about key distribution and not just confidentiality. The original implementation by Paulson [101] did not take into account the knowledge each peer acquired during the various stages of execution. Its main concerns regarded what the *Spy* was able to learn during the execution.

With the development of the idea that there ought to exist a formal guarantee that the protocol's goals are available to the peers in a realistic model [20] we have the necessity of applying the inductive relation *analz* to peers other than the *Spy*. This made the previous *spies* predicate obsolete and created the necessity of a broader function, now known as *knows*. The knows function represents how the knowledge of each peer — the Spy included — is expanded during the execution of the protocol and was the outcome of Bella's goal availability principle. Definition 25 shows our new specification for the *knows* function.

**Definition 25.** *Extended inductive relation representing Peer's knowledge under Multicast*

```
consts
  knows  :: "agent => event list => msg set"
primrec
  knows_Nil:   "knows A [] = initState A"
  knows_Cons:    "knows A (ev # evs) =
       (if A = Spy then
        (case ev of
           Says A' B X => insert X (knows Spy evs)
         | Multicast A' B XF =>  (XF ` set B) ∪  (knows Spy evs)
         | Gets A' X => knows Spy evs
         | Notes A' X  =>
              if A' ∈ bad then insert X (knows Spy evs)
              else knows Spy evs)
        else
        (case ev of
           Says A' B X =>
```

```
          if A'=A then insert X (knows A evs)
          else knows A evs
       | Multicast A' B XF =>
          if A'=A then  (XF ` set B) ∪  (knows A evs)
          else knows A evs
       | Gets A' X    =>
          if A'=A then insert X (knows A evs)
          else knows A evs
       | Notes A' X    =>
          if A'=A then insert X (knows A evs)
          else knows A evs))"
```

The inductive relation *knows* is now specified in 4 inductive cases in two steps. We start our inductive definition with the specification of the base case *knows_Nil*. In the base step *knows_Nil* the trace of events is empty and the knowledge of a peer is equal to its initial knowledge prior to the execution of the protocol. When we move to the step of a non-empty trace of events, we have two classes of peers and four cases each.

The *Spy* is able to learn differently than other peers. When he sees an event *Says* sending $X$ as a message, we extend his knowledge by inserting $X$ to his knowledge set. When he sees an event *Multicast* from a peer to a multicast group casting $XF$ we add to the knowledge of the *Spy* the image of the function $XF$ over the set of peers in the multicast group. When he sees a *Gets* event he learns nothing, because he already learnt it at the sending event. When a peer learns a message $X$ through the predicate *Notes*, the *Spy* will learn the message $X$ if the peer is corrupted.

The second class of peers regards all agents not colluding with *Spy*. When the peer originates an event *Says* to another peer casting $X$ as a message, we extend his knowledge by inserting $X$. If he is not the originator we do nothing. When the peer issues event *Multicast* to a multicast group casting $XF$, we add to his knowledge the image of the function $XF$ over the set of peers in the multicast group. When he *Gets* a message $X$ through the predicate *Gets* we insert $X$ to his knows set. When he learns a message $X$ through the predicate *Notes*, we insert $X$ to his knows set.

During the specification of the *event* datatype and the extension of the function *knows*, we realised the existence of some subtleties regarding the Multicast constructor and its reception, especially regarding knowledge gathering. Our initial design for the multicast event implementation conceived the introduction of a predicate specifically to deal with the reception of multicast messages.

Having a predicate *GetsMC* seemed attractive since a Multicast message conveys information regarding the knowledge other peers in the multicast group may have acquired. As an example, when we receive a Multicast message addressed to a group composed of us and another two peers, we don't just learn the contents of the message conveyed by the multicast, but also that the other two users may also have had their view of the message contents and learned what we considered the public components in the message. This idea can be extended even further, since the other two peers know that we may know the information conveyed by the message.

The reflexion of the inductive relation *knows* would add a new dimension to peers knowledge as the Multicast does to the event traces. We would stop having the linearity connecting trace expansion and peer knowledge acquisition, and add new possibilities of inferring other peers' knowledge. The extension of the inductive relation *knows* to deal with other peer's knowledge is very attractive to use with novel threat models. We can, for example, conjecture that this new peer's knowledge acquisition model can enable us to reason about the detection or not of retaliation attacks [11] in certain multicast scenarios.

We dropped the idea because gathering this information is difficult in practice because multicast is supposed to hide group composition. This is even more problematic in an environment where message reception cannot be guaranteed. The knowledge reflection would be just as a hint of other peer's knowledge and not a concrete fact. Another issue was that we did not want to break backward compatibility by changing the shape of the function *knows* and re-implement all the other affected definitions to accommodate this change.

With the new definition of the function *knows* to encompass a Multicast primitive we now can reason about knowledge acquired be the peers during the execution of Multicast based protocols, as well as design the necessary lemmas to make the specification usable. In section 3.2.3 we will describe the new lemmas that enable us to reason about the constructor *Multicast* as we did with the constructor *Says* in the Unicast scenario.

### 3.2.2 Extending Used set for Multicast Communications

Another important extension to the Inductive method by the addition of the Multicast predicate concerns to the extension of the *used* function which enablse us to reason about freshness. Freshness is a necessity for reasoning about the unicity of certain messages. It is also a key compositional property for reasoning about key distribution.

The *used* function forms the set of all message components that have already appeared in the event trace plus all the information all peers initiated the protocol run with. Definition 26 shows the extended version of the function *used*, now encompassing the Multicast primitive.

**Definition 26.** *Extended function that represents used parts of messages under Multicast environment*

```
consts
  used :: "event list => msg set"
primrec
 used_Nil:   "used []         = (UN B. parts (initState B))"
 used_Cons:  "used (ev # evs) =
               (case ev of
                Says A B X => parts {X} ∪ used evs
                | Multicast A B XF => parts (XF ` set B) ∪
                  used evs
                | Gets A X   => used evs
                | Notes A X  => parts {X} ∪ used evs)"
```

The *used* function is specified in two inductive steps. The first inductive step *used_Nil* is the base case, where our event trace is empty. It is defined by the union of the application of the function parts to the initial state of all peers. This definition allows us to consider as used any information.

The second inductive step regards the components used during trace construction. This inductive step has four cases, one for each event constructor present in the protocol construction model. The first case regards the primitive *Says*, where the application of the function parts on the message $X$ is joined with the used set of the remaining events. The second case regards the new addition of *Multicast*. In this case we apply the function parts to the image of function *XF* over the set of peers in the multicast group $B$ and join the result with the set of remaining events. The third case regards the *Gets* primitive, where no action is taken since the parts are already considered "used" when sent. Finally the fourth case regards the primitive *Notes*, where the content of what is being noted is passed through the function parts and then joined with the other used components.

The extension of the used function is key for adding Multicast support to the inductive method. Effectively the *used* set has been kept stable since the first implementations of the inductive method [101], since the additions made for encompassing message reception were only technical and did not change the set construction. Differently from *knows*, the *used* function would not yield any new property by the adding of a message reception primitive for Multicast messages, but the adding of the Multicast message primitive makes the set potentially bigger and more complex to reason about.

In the next subsection (§3.2.3) we will introduce the new lemmas needed to reason about the new additions made to the function knows and used.

## 3.2.3 Basic Lemmas

The basic lemmas for reasoning about the multicast primitives in the face of the extensions introduced to the *used* and *knows* specification are presented. The introduction of such lemmas comes from the existence of their Unicast counterparts, and is key for the usability of the inductive method under the new Multicast environment.

Usually the construction of proofs regarding secrecy requires a reference to the knowledge acquired during the execution of the protocol. It is also a necessity to supply the theorem prover with some properties to guide it toward the proof path we intend. The proof of Lemma 11 (*Multicast_implies_knows_Spy*) helps us to express that if a message *XF* was Multicasted to a multicast group $B$, then the image of the function *XF* over the set $B$ is a subset of the set of knowledge of the *Spy* in this trace.

**Lemma 11.** *Multicast_implies_knows_Spy*

```
Multicast A B XF ∈ set evs ⟹ XF ` set B ⊆ knows Spy evs
```

The proof of Lemma 11 (*Multicast_implies_knows_Spy*) is simple by induction over the set *evs* followed by the simplifier added with the case split argument of the *event* datatype.

Sometimes the argument of Lemma 11 (*Multicast_implies_knows_Spy*) needs to be applied to a specific recipient in the multicast group. This requirement yields Lemma 12 (*Multicast_implies_in_knows_Spy*). Lemma 12 (*Multicast_implies_in_knows_Spy*) states that if *XF* is the payload of a Multicast from agent $A$ to the multicast group $B$ and $C$ is and agent belonging to the multicast group, then the application of the function *XF* to $C$ is in the knowledge of the *Spy* in the trace *evs*.

**Lemma 12.** *Multicast_implies_in_knows_Spy*

```
⟦Multicast A B XF ∈ set evs; C ∈ set B⟧ ⟹ XF C ∈ knows Spy evs
```

Another case of the casting of a Multicast message into the network is the expansion of the knowledge set of the sending peer $A$. Lemma 13 (*Multicast_implies_knows*) states that if an agent $A$ casts a *Multicast* message *XF* to a multicast group $B$ then the image of the function *XF* over the set $B$ is a subset of the knowledge of peer $A$ in the event trace *evs*.

**Lemma 13.** *Multicast_implies_knows*

```
Multicast A B XF ∈ set evs ⟹ XF ‘ set B ⊆ knows A evs
```

Proof for Lemma 13 (*Multicast_implies_knows*) is similar to the one for Lemma 11 (*Multicast_implies_knows_Spy*).

An important simplification rule regards the equivalence between the extension of the trace and the extension of the Spy's knowledge. This equivalence is shown in Lemma 14 (*Know_Spy_Implies_Multicast*) which states that extending the knowledge of the *Spy* by adding a *Multicast* message to the trace of events is equivalent to the union of the image of the *XF* function over the set of peers in the multicast group with the knowledge the *Spy* already has. The proof of Lemma 14 (*Know_Spy_Implies_Multicast*) is done solely by the simplifier.

**Lemma 14.** *Know_Spy_Implies_Multicast*

```
knows Spy (Multicast A B XF # evs) = XF ‘ set B ∪ knows Spy evs
```

The knowledge set the *Spy* has regarding the trace before the casting of any *Multicast* message is a subset of the knowledge the *Spy* has after extending the trace by casting a *Multicast* message. Lemma 15 (*Knows_Spy_subset_knows_Spy_Multicast*) expresses the above property of the knows function. The existence of this fact to Isabelle/HOL is important because it can help on the simplification of message dependencies regarding the way subgoals are rewritten by the tool.

**Lemma 15.** *Knows_Spy_subset_knows_Spy_Multicast*

```
knows Spy evs ⊆ knows Spy (Multicast A B XF # evs)
```

Proving Lemma 15 (*Knows_Spy_subset_knows_Spy_Multicast*) is done automatically by Isabelle/HOL's auto tactic.

Lemma 16 (*knows_Multicast*) is a generalised version to peers other than the *Spy* of Lemma 11 (*Multicast_implies_knows_Spy*). In fact it represents the lawful acquisition of knowledge to any peer in the protocol execution when it casts a *Multicast* message. Lemma 15 (*Knows_Spy_subset_knows_Spy_Multicast*) states the equivalence between extending the knowledge of peer $A$ by extending trace adding a *Multicast* message with the application of the union between the image of the function *XF* over the set $B$ with the previous knowledge of peer $A$. It is proven by applying simplification only.

**Lemma 16.** *knows_Multicast*

```
knows A (Multicast A B XF # evs) = XF ‘ set B ∪ knows A evs
```

Lemma 17 (*knows_subset_knows_Multicast*) is a generalised version of Lemma 15 (*Knows_Spy_subset_knows_Spy_Multicast*) that establishes a fact for the legal execution of the protocols for all peers. This lemma is important for clearing the rewriting done by the theorem prover, and its proof is also done automatically by Isabelle/HOL.

**Lemma 17.** *knows_subset_knows_Multicast*

```
knows A evs ⊆ knows A (Multicast A' B X # evs)
```

The other technical lemmas that involve the extension of the inductive relation *knows* relate the knowledge of the *Spy* with the events that appeared on the trace. This is presented by Lemma 18. It states that if an element $X$ is in the set of knowledge of the *Spy* then there exists some $A$, some $B$, some $C$ and a function *XF* for that and a *Says* event mentioning $X$ is on the trace, or an event with the primitive *Multicast* mentioning the function *XF* is in the trace and the element $X$ is in the image of the function *XF* over the set $C$, or the element $X$ was Noted from the trace by an event *Notes* $X$ or the element $X$ was previously known by the *Spy*.

**Lemma 18.** *knows_Spy_implies_Says_Multicast_Notes_InitState*

```
X ∈ knows Spy evs ⟹
∃A B C XF.
   Says A B X ∈ set evs ∨
   Multicast A C XF ∈ set evs ∧ X ∈ XF ‘ set C ∨
   Notes A X ∈ set evs ∨ X ∈ initState Spy
```

The proof requires induction followed by using the simplifier extended with the case spilt for the datatype *event* followed by the classical reasoner from Isabelle/HOL.

The second technical lemma regarding the relation of knowledge of an element with trace inspection is applied to all peers other than the *Spy*. The introduction of such fact is done by Lemma 19 (*knows_implies_Says_Multicast_Gets_Notes_InitState*). It states that if an element $X$ is in the set of knowledge of peer $A$ over the trace *evs*, and this agent is not the *Spy*, it implies that there exist some $A$, some $B$ and a function *XF* so that a *Says* event mentioning $X$ is on the trace, or an event with the primitive *Multicast* mentioning the function *XF* is in the trace and the element $X$ is in the image of the function *XF* over the set $C$, or a Gets event mentioning $X$ is on the trace, or the element $X$ was Noted in the trace or the element $X$ was previously known by the peer $A$.

**Lemma 19.** *knows_implies_Says_Multicast_Gets_Notes_InitState*

```
⟦X ∈ knows A evs; A ≠ Spy⟧
⟹ ∃B C XF.
        Says A B X ∈ set evs ∨
        Multicast A C XF ∈ set evs ∧ X ∈ XF ‘ set C ∨
        Gets A X ∈ set evs ∨ Notes A X ∈ set evs ∨
        X ∈ initState A
```

The Proof is very similar to the one for 18.

Other technical lemmas are required for the extended *used* function. Lemma 20 (*Multicast_Implies_Used*), concerns the extension of the set *used* on the actual trace of events when a *Multicast* message is casted in a protocol. If we have a *Multicast* from agent $A$ to the multicast group $B$ with the payload *XF* then that the image of the function *XF* over the set $B$ is a subset of the set *used* in the current trace of events.

**Lemma 20.** *Multicast_Implies_Used*

```
Multicast A B XF ∈ set evs ⟹ XF ‘ set B ⊆ used evs
```

Proving Lemma 20 (*Multicast_Implies_Used*) requires the application of induction over *evs* followed by the classical reasoner augmented with the case split for the *event* datatype.

Lemma 21 (*Multicast_implies_in_Used*) states that if we have a *Multicast* from agent *A* to multicast group *B* with the payload *XF* and *C* is an agent of the multicast group *B* then the function *XF* of *C* is in the set *used* for the trace *evs*.

**Lemma 21.** *Multicast_implies_in_Used*

```
⟦Multicast A B XF ∈ set evs; C ∈ set B⟧ ⟹ XF C ∈ used evs
```

To achieve the proof of Lemma 21 (*Multicast_implies_in_Used*) we need to appeal to Lemma 20 (*Multicast_Implies_Used*) and to the reversibility of the set image operator which is not shown in this thesis.

Another important lemma is Lemma 22 (*used_Multicast*), which states the equivalence in the extension of the trace of events and the extension of the set of *used*. Proving Lemma 22 requires only a call to the simplifier.

**Lemma 22.** *used_Multicast*

```
used (Multicast A B XF # evs) = parts (XF ‘ set B) ∪ used evs
```

Leaving behind the more technical lemmas shown above and following to lemmas we normally appeal to when trying to state proofs concerning protocol objectives, we follow with the presentation of Lemma 23 (*Multicast_implies_in_parts_spies*). Although the name of this lemma still carries the old naming scheme from the *spies* function before the introduction of function *knows*, this lemma already takes into account the *knows* function and is very important. It is useful for proving goals regarding Regularity, Unicity and Authenticity.

Lemma 23 states that if the trace of events contains a *Multicast* event from agent *A* to multicast group *B* with the payload function *XF*, and *C* is and agent of the multicast group *B*, then the application of the payload function onto *C* is in the *parts* set for the *Spy's* knowledge over the trace.

**Lemma 23.** *Multicast_implies_in_parts_spies*

```
⟦Multicast A B XF ∈ set evs; C ∈ set B⟧
⟹ XF C ∈ parts (knows Spy evs)
```

The proof requires us to appeal to Lemma 12 (*Multicast_implies_in_knows_Spy*) and the injectiveness of *parts* .

To enable us to reason about Multicast and confidentiality goals we introduce facts based on the function *analz*. Lemma 24 (*Multicast_implies_analz_Spy*) is an important fact regarding the expansion of the *analz* set during the cast of a *Multicast* event. It states that if a *Multicast* event from agent *A* to the multicast group *B* with the payload function *XF* is in the trace of events, this implies that the image of the function *XF* over the set *B* is a subset of the set of message components the *Spy* can analyse from the trace.

**Lemma 24.** *Multicast_implies_analz_Spy*

```
Multicast A B XF ∈ set evs ⟹ XF ‘ set B ⊆ analz (knows Spy evs)
```

Facts that involve the operator *analz* proofs tend to be intrinsically complicated. We apply induction followed by the auto tactic augmenting the case split property of the *events* datatype. We are left with three subgoals. They regard the expansion of the knowledge set of the *Spy* after the unification with the image of function *XF* over the set *B*. Two of these subgoals are trivial and can be proved by appeal to the fact *analz_insert* which states that "$c \in G \Longrightarrow c \in$ *analz (insert a G )*" and the totality of function's images. The remaining subgoal requires an appeal the injectiveness of *analz*, the totality of function's images and to Lemma 11 (*Multicast_implies_knows_Spy*) .

Another important strategy within Isabelle/HOL used with the Inductive Method regards declaration of lemmas from previous facts. One Lemma produced by this technique is 25 (*Multicast_imp_parts_knows_Spy*). This lemma is important when we have to reason about the expansion of *parts*' image and the existence of atomic components of the *Multicast* message in the *parts* set. It is a useful fact for goals regarding Regularity and Unicity.

**Lemma 25.** *Multicast_imp_parts_knows_Spy*

```
⟦Multicast A B XF ∈ set evs; C ∈ set B;
XF C ∈ parts (knows Spy evs) ⟹ PROP W⟧ ⟹ PROP W
```

Our final extension to the inductive method to enable it to reason about *Mutlicast* events concerns to the expansion of the tactic *synth_analz_mono_contra*. This tactic is important for reasoning about theorems with the form $X \in$ *synth( analz( knows Spy evs))* $\Longrightarrow P$, and to clear certain subgoals that involve the fake cases. Our addition to the tactic was the inclusion of the *Multicast* processing in the function *synth_analz_mono_contra_tac*.

Following this explanation regarding the extensions needed to the inductive method to enable it to reason about Multicast communication primitives, we will be briefly discussing the re-interpretation of security goals and the new shape of the inductive method's infrastructure under a Multicast environment.

### 3.2.4 General Re-Interpretation of Security Goals under Multicast

Extending the Inductive method to accommodate a Multicast communication primitive, requires the re-interpretation of how the method works and how some security goals should be understood.

The first thing to bring to attention is the modification of the idea of trace of events. Prior to the introduction of the *Multicast* primitive, we had the idea of a linear trace and a linear expansion of the knows and used set. Following the ideas outlined by Bella [20] and shown on Figure 3.1, a generic trace is a list of any length that is the parameter for the modelling and verification of protocols properties.

Interpreting now the relation between trace expansion and the *knows* and *used* set expansions in the case of the trace shown by Figure 3.1, we can see the linearity in both expansions. For the sake of exemplifying, we know that the *knows* set for peer *A* is

```
Says Spy D {|Agent A, Nonce Nc|}
Notes Spy {|Nonce Nc, Key Kcb|}
Notes Spy {|Nonce Nc, Key Kcb|}
Says B C ( Crypt (priSK B) {| Nonce Nc, Key Kcb|})
Gets B (Nonce Nc)
Says Spy D (Nonce Na')
Says C B {| Agent C, Nonce Nc|}
Notes Spy ( Nonce Na' )
Says A B {| Agent A, Nonce Na'|}
Says A B {| Agent A, Nonce Na|}
```

Figure 3.1: Linear trace example from Bella[20]

initialised before any message is sent with all prior knowledge *A* had. This is generally composed by his own shared key and private keys as well as all the public keys (*initState* definition 13). After the sending of the first message onto the trace, by the definition of knows, the knowledge of peer *A* will be extended by inserting the payload of the message to its knowledge set. And the same will happen for any *Notes* event.

After the extensions proposed by Bella [20], we see the knowledge set of a peer also being extended by message reception event Gets. In this scenario, the peer receiving a message from the network will insert its payload onto his own knowledge set. A very similar procedure happens to the extension of the *used* set. To every message send onto the trace we expect a linear expansion of the sets *knows* and *used* by the size of the payload of the *Says* event.

We look closer to the relation happening in a one-to-many communication style we see that this linearity is lost. We cast a Multicast event in the trace we are not extending the knows and used set by exactly the payload of the message, but by the application of the payload function over the list of agents in the multicast group. This inherently changes the trace construction since we also do not have this conveying of knowledge between two peers only. The inductive method is capable of coping with this immense potential growth in the size of the trace information, which clearly corroborates our choice in using it as the testbed for our implementation efforts.

Another important re-interpretation concerns the goal of Secrecy, due to the leakage of information to peers by the usage of side-channels inherent to a Multicast implementation. As mentioned earlier, with the exception of the multicast implementation being used in an Anycast mode of operation we can argue that there is a side channel leaking information that could be captured by the knowledge representation within the method. Although the method is not prepared today to make use of this information, with the expansion of it to encompass newer threat models, this can be important to represent knowledge that could help us to protect or prepare better retaliation attacks.

## 3.3 Verification of a Classic Unicast Protocol

One of the main challenges of implementing Multicast support to the Inductive Method was making it capable of covering the extreme cases. Following the idea that Unicast and

Broadcast are the extremes of a Multicast communication, in this section we show how a Unicast protocol can have its properties verified using our Multicast implementation in Unicast mode of operation. Unicast is achieved by sending a Multicast to a multicast group of size one. We chose to do this demonstration using a classical protocol; Needham-Schroeder Shared Key [93] .

The choice of Needham-Schroeder Shared Key protocol is justified by the fact it has well known security goals and its verification was already very well covered elsewhere [101]. Its authentication and key distribution goals made it well suited for our demonstration aims. Its proofs achieve key distribution and authentication of both peers with the assistance of a trusted third party produce enough details to measure the effort needed to use Multicast as the main event model in the Inductive Method.

In the next subsections we will cover the modifications in the Needham-Schroeder Shared Key protocol to make it verifiable under the Multicast environment and the choices made in this re-implementation (§3.3.1). Following the description of the changes (§3.3.2) we will cover the verification process(§3.3.3), focusing on the challenges introduced in the proof methods and how this impacts the outcomes and effort for the protocol verification process.

## 3.3.1 Revisiting Needham-Schroeder Shared Key

We revisit the Needham-Schroeder Shared Key protocol to verify it under the Multicast model. By doing this, we will be creating a new protocol variant called Needham-Schroeder Shared Key Multicast Mode. This is achieved by substituting all the Unicast primitives by Unreliable Multicast ones, thus, enabling the protocol to use our Multicast implementation in the process.

As the unicast version of the protocol has been verified, no major attack was expected to be found. Our main goal was to show that a Multicast implementation can be used for Unicast, giving the basis to suggest it as the default implementation for the Inductive Method. Our objective is to measure the new complexity introduced in the proof construction process.

The classical Unicast protocol is then transformed to a Multicast variant as shown in Figure 3.2.

| | | | | | |
|---|---|---|---|---|---|
| 1. | $A$ | $\overset{U}{\rightsquigarrow}$ | $Server$ | : | $\lambda C.\{\vert A, B, Na\vert\}$ |
| 2. | $Server$ | $\overset{U}{\rightsquigarrow}$ | $A$ | : | $\lambda C.\{\vert Na, B, K_{AB}, \{\vert K_{AB}, A\vert\}_{sK_B}\vert\}_{sK_A}$ |
| 3. | $A$ | $\overset{U}{\rightsquigarrow}$ | $B$ | : | $\lambda C.\{\vert K_{AB}, A\vert\}_{sK_B}$ |
| 4. | $B$ | $\overset{U}{\rightsquigarrow}$ | $A$ | : | $\lambda C.\{\vert Nb\vert\}_{K_{AB}}$ |
| 4. | $A$ | $\overset{U}{\rightsquigarrow}$ | $B$ | : | $\lambda C.\{\vert Nb - 1\vert\}_{K_{AB}}$ |

Figure 3.2: Needham-Schroeder Shared Key Multicast Implementation

Obvious choices in the translation were made, such as the usage of an Unreliable Multicast, since the previous protocol verification scenario had the same constraint of unreliability of Unicast. Note that we do not use functions to address the keys being used to encrypt traffic, as it would be expected in a true multiple receiver Multicast protocol.

This is because there will be only one encryption. The choice was done for the sake of fidelity with the real implementation of the protocol. The usage of these functions would only impact in the proof strategy. The main change would be to use more aggressive destruction rules for function congruence.

### 3.3.2 Needham-Schroeder Shared Key Multicast Modelling

The verification of the Needham-Schroeder Shared Key Multicast consists first of the creation of a specification in Isabelle/HOL that represents the protocol defined in the Figure 3.2. To start a constant *ns_shared_MC* is declared as an inductive set of lists of events which represents the formal protocol model. This inductive set setup is shown in Definition 27.

**Definition 27.** *Inductive Definition of Needham-Schroeder Shared Key Distribution Protocol using Multicast Primitive - setup*

```
inductive_set ns_shared_MC :: "event list set"
 where
```

We define the empty trace by the rule *Nil*, which sets the base of the induction.

The *Spy*'s illegal activity is formalised by the rule *FakeMC*, which includes any forging the *Spy* is able to perform from the knowledge acquired from the trace of events so far. Rule *FakeMC* is a variant from the rule *Fake* from Unicast verification process. It gives the *Spy* the power to multicast any message with components he learnt from traffic analysis, to any multicast group present in the execution of the protocol. The isabelle/HOL implementation of the rules *Nil* and *FakeMC* are shown in Definition 28.

**Definition 28.** *Inductive Definition of Needham-Schroeder Shared Key Distribution Protocol using Multicast Primitive - Empty set and Fake Rules*

```
  Nil:  "[] ∈ ns_shared_MC"

| Fakemc: "⟦ evsfmc ∈ ns_shared_MC;
          XF ∈ synth(analz(spies evsfmc))⟧
          ⟹ Multicast Spy multicast_group (λC. XF ) #
          evsfmc ∈ ns_shared_MC"
```

The protocol then starts with a phase composed by two messages where the agent $A$ will request the trusted third party *Server* to initialise a communication with the agent $B$. During this phase, the *Server* will generate a session key to be used by agents $A$ and $B$ and will give to $A$ the means to proceed to the next phase.

To address this phase, our model is extended initially by the rule *NS1*. The only precondition of the rule *NS1* is that the session identifier *NA* was never used in the set of traces of events, avoiding then the collision with a pre-existent session. If this precondition is met we extend the trace of events by adding a *Multicast* message from the agent $A$ to the multicast group composed of the agent *Server*. The message carries agent's $A$ identity, agent's $B$ identity, and the session identifier *NA*.

The next extension of the model is done by rule *NS2*. The preconditions of this rule state that if the trace of events contains a message with the syntax of the message added by the rule *NS1*, from any agent and addressed to the *Server* and the session key *KAB* is indeed a session key and was never used before in the set of traces of events, we can extend the actual trace. We then add to the trace of events a *Multicast* from the *Server* to the multicast group composed solely of agent $A$, containing the identity of agent $A$, the identity of agent $B$, the session identifier *NA* and a certificate for $A$ to forward to $B$. This certificate is composed by the session key $K_{AB}$ and the identity of $A$, encrypted with the long-term shared key of the agent $B$. The whole message Multicasted by the *Server* to agent $A$ is encrypted under the long-term shared key of agent $A$.

The Isabelle/HOL implementation of rule *NS1* and rule *NS2* can be seen on Definition 29.

**Definition 29.** *Inductive Definition of Needham-Schroeder Shared Key Distribution Protocol using Multicast Primitive - Rule NS1 and Rule NS2*

```
| NS1:  "⟦evs1 ∈ ns_shared_MC;  Nonce NA ∉ used evs1 ⟧
         ⟹ Multicast A [Server] (λC.
         ⦃Agent A, Agent B, Nonce NA⦄) # evs1 ∈ ns_shared_MC"

| NS2:  "⟦evs2 ∈ ns_shared_MC; Key KAB ∉ used evs2; KAB ∈ symKeys;
          Multicast A' [Server] (λC.
          ⦃Agent A, Agent B, Nonce NA⦄ ) ∈ set evs2⟧
         ⟹ Multicast Server [A] (λC.
               (Crypt (shrK A)
                  ⦃Nonce NA, Agent B, Key KAB,
                     (Crypt (shrK B) ⦃Key KAB, Agent A⦄)⦄) )
               # evs2 ∈ ns_shared_MC"
```

After agent $A$ received from the *Server* the certificate and the session key necessary to proceed the communication establishment, she forwards the certificate to agent $B$. Upon receiving the certificate, agent $B$ recognises it because it was encrypted with the key he shares with the *Server*.

To test agent's $A$ aliveness and possession of the session key, agent $B$ sends agent $A$ a message containing a freshness value encrypted with the session key. Upon receiving $B$'s message, agent $A$ decrypts it with the session key. To prove the possession of the key and his aliveness, agent $A$ sends agent $B$ a message with the freshness value modified in an expected way and encrypted under the session key.

To cover the above steps our model is extended by the rule *NS3*. This rule has as preconditions that the trace of events contains a *Multicast* event sent by some agent $S$ to agent $A$ with the syntax of rule *NS2*, and that the trace of events contains a *Multicast* event from agent $A$ to the *Server* with the syntax of rule *NS1*. If these preconditions are met we can extend the trace with the *Multicast* event from agent $A$ to the multicast group represented solely by agent $B$ sending the certificate agent $A$ received from agent $S$. Deliberately we use agent $S$ instead of agent *Server*, because we cannot know the message indeed came from agent *Server*.

Our next extension to the model is rule *NS4*. The preconditions of rule *NS4* are that the freshness value *NB* was never used in the set of traces before, and the key $K$ received inside the certificate is indeed a session key. We also have a precondition that the trace of events contains a *Multicast* from some agent $A$ to some multicast group containing $B$

with a session key *K* and the identity of agent *A* encrypted under the shared key between agent *B* and the *Server*. If the preconditions are met we can extend the trace of events with a *Multicast* event from agent *B* to the group solely represented by agent *A* containing the freshness value *NB* encrypted with the session key *K*.

The model is then extended to the last message of the protocol by rule *NS5*. The preconditions of rule *NS5* are that key *K* is a session key and the trace of events contain a *Multicast* from an agent *B'* to *A* with the syntax of rule *NS4* and a *Multicast* message from an agent *S* to a multicast group containing agent *A* with the syntax of rule *NS2*. If the preconditions are met, we can extend the trace of events by adding a *Multicast* event from agent *A* to the multicast group represented by agent *B* of the agreed modification of the freshness value *NB* encrypted by the session key *K*.

The Isabelle/HOL implementation of rule *NS3*, rule *NS4* and rule *NS5* can be seen on Definition 30.

**Definition 30.** *Inductive Definition of Needham-Schroeder Shared Key Distribution Protocol using Multicast Primitive - Rule NS3, NS4 and NS5*

```
| NS3:  "⟦evs3 ∈ ns_shared_MC;  A ≠ Server;
          Multicast S [A] (λC. (Crypt (shrK A)
          ⦃Nonce NA, Agent B, Key K, X⦄) ) ∈ set evs3;
          Multicast A [Server] (λC.
          ⦃Agent A, Agent B, Nonce NA⦄ ) ∈ set evs3⟧
        ⟹ Multicast A [B] (λC. X ) # evs3 ∈ ns_shared_MC"

| NS4:  "⟦evs4 ∈ ns_shared_MC; Nonce NB ∉ used evs4; K ∈ symKeys;
          Multicast A' [B] (λC. (Crypt (shrK B)⦃Key K,Agent A⦄))
          ∈ set evs4⟧
        ⟹ Multicast B [A] (λC. (Crypt K (Nonce NB)))
        # evs4 ∈ ns_shared_MC"

| NS5:  "⟦evs5 ∈ ns_shared_MC;  K ∈ symKeys;
          Multicast B' [A] (λC. (Crypt K (Nonce NB))) ∈ set evs5;
          Multicast S [A] (λC. (Crypt (shrK A)
          ⦃Nonce NA, Agent B, Key K, X⦄) )
            ∈ set evs5⟧
        ⟹ Multicast A [B] (λC. (Crypt K ⦃Nonce NB, Nonce NB⦄))
        # evs5 ∈ ns_shared_MC"
```

Finally we add to our definition the possibility of having session keys being compromised by the *Spy*. Although this is not part of the standard protocol it is added to our model to be able to show that the loss of control in one session does not represent problems to other concurrent sessions. The preconditions of rule *Oops* are that a *Multicast* from an agent *B* to to a multicast group *A* is on the trace with the syntax of the rule *NS4* and a *Multicast* from the *Server* to the same multicast group *A* with the syntax of rule *NS2* are on the set of traces of events. If the preconditions are met, we add to the knowledge of the *Spy* through a *Notes* event, the knowledge of the freshness value *NA*, the freshness value *NB* and the session key *K*.

The Isabelle/HOL implementation of rule *Oops* can be seen on Definition 31.

**Definition 31.** *Inductive Definition of Needham-Schroeder Shared Key Distribution Protocol using Multicast Primitive - Rule Oops*

```
| Oops: "⟦evso ∈ ns_shared_MC;
          Multicast B [A] (λC. (Crypt K (Nonce NB)) ) ∈ set evso;
          Multicast Server [A] (λC. (Crypt (shrK A)
          ⦃Nonce NA, Agent B, Key K, X⦄) ) ∈ set evso⟧
         ⟹ Notes Spy ⦃Nonce NA, Nonce NB, Key K⦄
          # evso ∈ ns_shared_MC"
```

We should make clear that the protocol steps are represented by rules *NS1*, *NS2*, *NS4*, *NS4* and *NS5*.

In the next subsection, we will be looking over the verification process of the protocol modelled here, and at the end of this chapter we will be analysing the outcomes of our re-implementation.

### 3.3.3 Needham-Schroeder Shared Key Multicast Verification

In this subsection we will present the main guarantees proven for the Needham-Schroeder Shared Key Multicast Protocol. Our main goal was not to find new attacks on the modified version of the protocols, but to measure the effort needed to rebuild the tactics for proving the new variant correct.

The term *evs* will always stand for a generic trace of the formal protocol model specified in the previous section. We will be presenting the proof in a forward style, covering initially subgoals of Regularity and Reliability, followed by Authenticity, Unicity, Confidentiality and and Authentication. Finally we will show some of the newly introduced lemmas that enabled us to reason about the previous properties using the Multicast variant of the events model.

#### 3.3.3.1 Regularity

An important regularity property present in the protocols is that the long term keys shared between the *Server* and the general population of agents cannot appear on the traffic. If this lemma appears, the protocol is not being followed and the player whose key appeared is corrupted by the *Spy*. This is expressed by Lemma 26 (Spy_Analz_ShrK).

**Lemma 26.** *Spy_Analz_ShrK*

```
evs ∈ ns_shared_MC ⟹ (Key (shrK A) ∈ analz (knows Spy evs)) =
(A ∈ bad)
```

Lemma 26 (Spy_Analz_ShrK) is easily provable. We will appeal to Lemma 27 to enable us to reason about the encrypted part of the rule *NS3* and to Lemma 28 to reason about the accidental loss of session keys. An important difference in terms of proof strategy in this lemma is that we need to explicitly appeal to Lemma 28(Oops_parts_spies) to help the classical reasoner to solve it.

**Lemma 27.** *NS3_msg_in_parts_spies*

```
Multicast S [A] (λC. Crypt (shrK A) ⦃N, B, K, X⦄) ∈ set evs
⟹ X ∈ parts (knows Spy evs)
```

**Lemma 28.** *Oops_parts_spies*

```
Multicast Server [A] (λC. Crypt (shrK A) {|NA, B, K, X|}) ∈ set evs
⟹ K ∈ parts (knows Spy evs)
```

Lemmas 27 (NS3_msg_in_parts_spies) and 28 (Oops_parts_spies) now need the explicit mention of Lemma 23 (Multicast_implies_in_parts_spies) instead of the sole use of the classical reasoner, what is not essentially difficult to do.

In all three cases of Regularity, proof effort was very similar to doing the proof using the Unicast implementation, and is easily sorted out by using the automation embedded in Isabelle/HOL, especially the proof assistant Sledgehammer.

#### 3.3.3.2 Reliability

As usual in key distribution protocols based on shared key cryptography, a relevant Lemma regards the reliability of the *Server* since this property is for the distribution of session keys. It is expressed by the Lemma 29 (Says_Server_Message_Form).

**Lemma 29.** *Says_Server_Message_Form*

```
⟦Multicast Server [A] (λC. Crypt K' {|N, Agent B, Key K, X|})
∈ set evs;  evs ∈ ns_shared_MC⟧
⟹ K ∉ range shrK ∧
X = Crypt (shrK B) {|Key K, Agent A|} ∧ K' = shrK A
```

The proof method for Lemma 29 (Says_Server_Message_Form) is very close to its Unicast counterpart. We first prepare and apply induction them we call the simplifier and the classical reasoner. The main difference in the Multicast version of the proof is that we need to give the classical reasoner knowledge of function congruence, and apply it aggressively to converge to the proof.

Another important reliability lemma regards the tamper-proof evidence that the certificate agent *A* receives from the *Server* which he will forward to agent *B*. Lemma 30 (cert_A_Form) shows us this property.

**Lemma 30.** *cert_A_Form*

```
⟦Crypt (shrK A) {|NA, Agent B, Key K, X|} ∈ parts (knows Spy evs);
A ∉ bad;  evs ∈ ns_shared_MC⟧
⟹ K ∉ range shrK ∧ X = Crypt (shrK B) {|Key K, Agent A|}
```

Lemma 30 (cert_A_Form) is proven in the very same form its Unicast counterpart is. We give to the classical reasoner Lemma 29 (Says_Server_Message_Form) and Lemma 31 (A_trusts_NS2).

As with Regularity Lemmas, proving Reliability Lemmas using the Multicast events theory is straigh forward. The usual main modification is the usage of function congruence needed to reason about equalities in terms of the function present in each Multicast event.

#### 3.3.3.3 Authenticity

Proving authenticity goals also means to prove a message is trusted by the receiving peer. So, each sent message we should have a Lemma that stated its authenticity. This is true in the Needham-Schroeder Shared Key Multicast protocol, with the exception of Message One. This happens due to the fact message one does not use any sort of encryption, and thus no authentication mechanism.

Lemma 31 (A_trusts_NS2) shows us that if a message with the syntax of Message two was sent in the trace, it must have originated with the Agent Server.

**Lemma 31.** *A_trusts_NS2*

```
⟦Crypt (shrK A) ⦃NA, Agent B, Key K, X⦄ ∈ parts (knows Spy evs);
A ∉ bad;  evs ∈ ns_shared_MC⟧
⟹ Multicast Server [A] (λC.Crypt (shrK A)⦃NA,Agent B,Key K,X ⦄)
    ∈ set evs
```

The proof is similar to its Unicast counterpart. We start both by preparing and applying induction, followed by the appeal to Lemma 27 (NS3_msg_in_parts_spies) to enable us to reason about the encrypted part of message three. Lemma 31 (A_trusts_NS2) now requires us to give the classical reasoner knowledge of function congruence, and apply it aggressively to reason about the Multicast messages' function equalities. We are then left with rule *FakeMC*, which requires the appeal to Lemma 26 (Spy_Analz_ShrK).

Lemma 32 (B_trusts_NS3) produces similar guaranties as Lemma 31 (A_trusts_NS2). In this case we assert that if a message component with the syntax of the certificate forwarded in rule *NS3* is in the trace, than that certificate originated with the *Server*.

**Lemma 32.** *B_trusts_NS3*

```
⟦Crypt (shrK B) ⦃Key K, Agent A⦄ ∈ parts (knows Spy evs);
B ∉ bad;  evs ∈ ns_shared_MC⟧
⟹ ∃NA. Multicast Server [A]
          (λC. Crypt (shrK A)
               ⦃NA,Agent B,Key K,Crypt(shrK B)⦃Key K,Agent A⦄⦄)
         ∈ set evs
```

The proof is similar to proving Lemma 31 (A_trusts_NS2). We start both by preparing and applying induction, followed by the appeal to Lemma 27 (NS3_msg_in_parts_spies). We then apply the simplifier followed by the classical reasoner and we are then left with rule *FakeMC*, which requires the appeal to Lemma 26 (Spy_Analz_ShrK).

Lemma 33 (A_trusts_NS4) regards the authenticity of Message *NS4* to the receiving peer *A*. It states that if a message component with the syntax of rule *NS4* is in the trace, a message component with the syntax of rule *NS2* is in the trace, the confidential information was not leaked by rule *Oops*, and both the sender and the receiver of the message four are not colluding with the *Spy*, then message four indeed was generated by agent *B*.

**Lemma 33.** *A_trusts_NS4*

```
⟦Crypt K (Nonce NB) ∈ parts (knows Spy evs);
 Crypt (shrK A) {|NA, Agent B, Key K, X|} ∈ parts (knows Spy evs);
 ∀NB. Notes Spy {|NA, NB, Key K|} ∉ set evs; A ∉ bad; B ∉ bad;
 evs ∈ ns_shared_MC⟧
⟹ Multicast B [A] (λC. Crypt K (Nonce NB)) ∈ set evs
```

The proof is closely related to its Unicast counterpart. The proof appeals to Lemma 31 (A_trusts_NS2) and to the fact the *Spy* cannot see the content of encrypted messages (Spy_not_see_encrypted_key)[1]

We start both by preparing and applying induction, followed by the appeal to Lemma 27 (NS3_msg_in_parts_spies). We then apply the simplifier followed by the classical reasoner. The remaining cases are sorted out by the usage of the *analz_mono_contra* tactic and the simplifier and classical reasoner appealing aggressively to the function congruence lemma. We are then left with the subgoals yielded by rule *NS4*, which can be proven by appealing to Lemma 35 (unique_session_keys), Lemma 32 (B_trusts_NS3) and Lemma 11 (Multicast_implies_in_parts_spies).

Comparing the proof of Lemma 33 (A_trusts_NS4) with its Unicast counterpart, we can see some complication arising due to the function used in the Multicast implementation. But all the cases that were not straightforward could easily be solved with the help of the proof assistant Sledgehammer.

Finally Lemma 34 (B_trusts_NS5) regards the authentication of agent *A* as the sender of message five. If the message yielded by rule *NS5* appears on the trace, the message yielded by rule *NS3* appears on the trace, no keys were leaked to the *Spy* through the rule *Oops*, and both sender and receiver are not colluding with the *Spy*, then the message yielded by rule *NS5* in fact originated with the agent *B*.

**Lemma 34.** *B_trusts_NS5*

```
⟦Crypt K {|Nonce NB, Nonce NB|} ∈ parts (knows Spy evs);
 Crypt (shrK B) {|Key K, Agent A|} ∈ parts (knows Spy evs);
 ∀NA NB. Notes Spy {|NA, NB, Key K|} ∉ set evs; A ∉ bad; B ∉ bad;
 evs ∈ ns_shared_MC⟧
⟹ Multicast A [B] (λC. Crypt K {|Nonce NB, Nonce NB|}) ∈ set evs
```

The proof appeals to Lemma 32 (B_trusts_NS3) and the fact that the *Spy* cannot see the session keys encrypted under long term keys (*Spy_not_see_encrypted_key*).

The proof relies on an intermediate lemma which is proven by preparing and applying induction, followed by the appeal to Lemma 27 (NS3_msg_in_parts_spies) as usual. The proof is very similar to the proof of Lemma 33 (A_trusts_NS4), leaving us with the subgoal yielded by the rule *NS5*.

The subgoal yielded by rule *NS5*, proved to be difficult to solve in comparison to its Unicast version. We start appealing to Lemma 11 (Multicast_implies_in_parts_spies) and then to the classical reasoner applying aggressive function congruence. This will yield two new subgoals that are solved similarly. We divide each subgoal by the case of the sender agent being colluding with the *Spy*, and then prove the first case appealing to the fact that if the *Spy* is not able to see session keys encrypted under long-term shared

[1]This lemma is part of the basic theories of the inductive method and is not shown in here. Its proof is available in the Isabelle/HOL distribution files

keys. Then to the agent to whom the long-term shared key belongs is colluding with the *Spy* (Crypt_Spy_analz_bad)[2]. The second case is solved as the Unicast counterpart by appealing to Lemma 31 (A_trusts_NS2) and Lemma 35 (unique_session_keys).

Verifying authenticity properties using the Multicast event implementation added some complications to proofs. In general these complications arise due to the way Multicast is implemented using functions, and also because in some cases the classical reasoner was not able to perform without the correct guidance.

#### 3.3.3.4 Unicity

Unicity properties concern the creation and use of fresh values and their correct use by the agents executing the protocol. They binds a fresh value to the message that originated it and to other components present in this message.

In the Needham-Schroeder Shared Key Multicast Protocol we have a lemma for the unicity of sessions keys distributed by the *Server*. Lemma 35 (unique_session_keys) states that if a session key $K$ is Multicasted by the *Server*, it is unique and bound to the other values of the message. Clearly, a key $K$ cannot be bound to two different sessions or freshness values $NA$.

**Lemma 35.** *unique_session_keys*

```
⟦Multicast Server [A] (λC. Crypt (shrK A)
                           ⦃NA, Agent B, Key K, X⦄) ∈ set evs;
 Multicast Server [A'] (λC. Crypt (shrK A')
                           ⦃NA', Agent B', Key K, X'⦄) ∈ set evs;
 evs ∈ ns_shared_MC⟧
⟹ A = A' ∧ NA = NA' ∧ B = B' ∧ X = X'
```

The proof is very similar to its Unicast counterpart. We start by preparing and applying induction, followed by a call to the simplifier and the classical reasoner, applying function congruence. The main difference in the proofs is that to prove the subgoal yielded by the rule *NS2* we need to explicitly appeal to Lemma 27 (NS3_msg_in_parts_spies).

Proving unicity under the Multicast event model is straightforward and easily achievable, requiring minor guidance to the classical reasoner which was implicit in the Unicast variant.

#### 3.3.3.5 Confidentiality

Confidentiality of certain components is necessary for the protocol to achieve its goals. The Needham-Schroeder Shared Key Multicast Protocol has an important confidentiality lemma regarding the secrecy of the sessions keys distributed.

Lemma 36 (secrecy_lemma) states that if the trace of events is extended by the message yielded by rule *NS2*, and both the peers sharing the session key are not colluding with the *Spy*, then the non-existence of a leak of information by the rule *Oops* implies that the *Spy* does not know the session key $K$.

**Lemma 36.** *secrecy_lemma*

[2]This lemma is part of the basic theories of the inductive method and is not shown in here. Its proof is available in the Isabelle/HOL distribution files

```
⟦Multicast Server [A]
  (λC. Crypt (shrK A) ⦃NA, Agent B, Key K, Crypt (shrK B)
                            ⦃Key K, Agent A⦄⦄) ∈ set evs;
 A ∉ bad; B ∉ bad; evs ∈ ns_shared_MC⟧
⟹ (∀NB. Notes Spy ⦃NA, NB, Key K⦄ ∉ set evs) ⟶
     Key K ∉ analz (knows Spy evs)
```

To prove the secrecy argument for the session keys (Lemma 36 (secrecy_lemma)) we do the usual induction preparation and induction application followed by the appeal to Reliability arguments, such as Lemma 29 (Says_Server_Message_Form). For the other subgoals, except the yielded by rule *NS3*, we appeal to Lemma 27 (NS3_msg_in_parts_spies) and Lemma 35 (unique_session_keys).

The proof for the subgoal yielded by rule *NS3* requires Lemma 31 (A_trusts_NS2) and the fact that the *Spy* is able to see session keys encrypted under shared keys, and that the agent to whom the shared key belongs is colluding with the *Spy*. This is followed by the appeal to Lemma 35 (unique_session_keys) and also to Lemma 11 (Multicast_implies_in_parts_spies).

The proof of Confidentiality properties using the Multicast event model is considerably more difficult that its Unicast counterpart. This happens because the classical reasoner is able to sort out small issues using its internal rules. But, nevertheless, the use of the Multicast event model requires clear understanding of some steps, such as the necessity of appealing to Lemma 28 (Oops_parts_spies) to prove the subgoal yielded by rule *NS2*.

#### 3.3.3.6 Authentication and Key Distribution

The ultimate goal of the Needham-Schroeder Shared Key Multicast Protocol is to allow key distribution and authentication to the agents involved in protocol execution, even in the presence of a powerful *Spy*.

To be able to establish mutual weak agreement with the protocols we need to trace back the originator of the authenticator. To be able to do so, we need to add a new predicate called *Issues*. It was already shown in the previous chapter, but this is a Multicast variant.

**Definition 32.** *Definition of the predicate Issues*

```
A Issues B with X on evs ≡
∃Y. Multicast A [B] (λC. Y) ∈ set evs ∧
     X ∈ parts {Y} ∧
     X ∉ parts
(knows Spy (takeWhile (λz.z ≠ Multicast A [B] (λC.Y)) (rev evs)))
```

As shown in Definition 32, we can assert the $A$ Issues $B$ with $X$ if $X$ is part of a message multicasted from $A$ to multicast group $B$ and $X$ never appeared in the trace before this event. The implementation details of Issues requires the creation of a series of technical lemmas to deal with the transposition of the trace with the operation takeWhile. These lemmas are not discussed here.

The authentication and key distribution from agent $A$ to agent $B$ is stated by Lemma 37 (A_authenticates_and_keydist_to_B). If the freshness value *NB* encrypted by the session key $K$ appears on the trace, the contents of message two encrypted under the long-term shared key between agent $A$ and *Server* also appears on the trace, the key $K$ is not available to the *Spy* and *A* and *B* are not colluding with the *Spy*, then *B* is the true originator of *NB* encrypted by the session key *K* on the trace.

**Lemma 37.** *A_authenticates_and_keydist_to_B*

```
⟦Crypt K (Nonce NB) ∈ parts (knows Spy evs);
 Crypt (shrK A) {|NA, Agent B, Key K, X|} ∈ parts (knows Spy evs);
 Key K ∉ analz (knows Spy evs);
 A ∉ bad; B ∉ bad; evs ∈ ns_shared_MC⟧
⟹ B Issues A with Crypt K (Nonce NB) on evs
```

Proving Lemma 37 (A_authenticates_and_keydist_to_B) requires the appeal to Lemma 33 (A_trusts_NS4) and Lemma 31 (A_trusts_NS2), since the authenticity of what agent *A* receives from the trace is important for asserting the authentication of the key distribution.

Treating the predicate *Issues* requires giving to the simplifier the lemmas regarding transposition of the trace and giving the classical reasoner the function congruence lemma. We are left with the subgoals yielded by the rule *NS3* and rule *NS4*. The subgoal yielded by rule *NS3* is solved by appealing to Lemma 11 (Multicast_implies_in_parts_spies) and to Lemma 30 (cert_A_Form). The other subgoal needs a special application of the transposition lemmas not covered here.

Mutual weak agreement demands proving the same guarantees we proved for *A* in Lemma 37 (A_authenticates_and_keydist_to_B) now to *B*. This is shown in Lemma 38 (B_authenticates_and_keydist_to_A). It states that if the content of message five appears on the trace, the certificate issued by the *Server* to *B* is on the trace, the key *K* is not available to the *Spy* and agents *A* and *B* are not colluding with the *Spy*, then *A* is the true originator of the message five on the trace.

**Lemma 38.** *B_authenticates_and_keydist_to_A*

```
⟦Crypt K {|Nonce NB, Nonce NB|} ∈ parts (knows Spy evs);
 Crypt (shrK B) {|Key K, Agent A|} ∈ parts (knows Spy evs);
 Key K ∉ analz (knows Spy evs);
 A ∉ bad; B ∉ bad; evs ∈ ns_shared_MC⟧
⟹ A Issues B with Crypt K {|Nonce NB, Nonce NB|} on evs
```

To prove Lemma 38 (B_authenticates_and_keydist_to_A) we have to appeal to Lemma 34 (B_trusts_NS5) and Lemma 32 (B_trusts_NS3). Similarly to the previous case with Lemma 37 (A_authenticates_and_keydist_to_B), we need to guarantee the authenticity of what agent *B* sees on the trace to prove the key distribution goal.

The predicate *issues* requires us also to give to the simplifier the lemmas regarding transposition of the trace and giving the classical reasoner the function congruence lemma. To sort out the remaining subgoals we have to appeal to Lemma 11 (Multicast_implies_in_parts_spies) and to Lemma 30 (cert_A_Form). We are then left with the subgoal yielded by the rule *NS5*. This subgoal appeals to a special version of the trace's transposition lemma and a lemma not covered here called *A_trusts_NS5,* which is similar to Lemma 34 (B_trusts_NS5) but with agent's *A* guarantees.

The proofs regarding Lemma 37 (A_authenticates_and_keydist_to_B) and Lemma 38 (B_authenticates_and_keydist_to_A) are very similar to their Unicast counterparts and did not demand the construction of any new proof strategy, except the usage of the function congruence rule.

### 3.3.4 Considerations of Needham-Schroeder Shared Key Multicast Verification

The Verification of the Needham-Schroeder Shared Key Multicast Protocol succeeded in proving the same goals as the classical Needham-Schroeder Shared Key protocol, corroborating the claim that the new construction of the Multicast event model is sufficient for the verification of Unicast protocols.

The objective of this revisit to an already verified protocol is twofold. First was the above stated claim of coverage of our new specification, and secondly there was the intent of measuring the effort introduced in the adoption of a general Multicast model for representing Unicast communications in the verification process of Security protocols. The idea was to corroborate the claims that Unicast and Broadcast are extremes for Multicast at the communication level, and to show our implementation was capable of representing such detail.

Taking into consideration the verification process shown above, we can see that for the verification of Regularity, Reliability, Unicity and Authentication/Key Distribution goals in general terms did not add any new proof strategy effort, except the fact of adding the required function congruence lemma needed for sorting out the equalities yielded by the function on the Multicast events.

Authentication goals generally required a better understanding of the proof strategy and a fierce guidance of the theorem prover to achieve the proofs. This happens due to the way Multicast events are implemented using functions to derive the view of each recipient of the multicast group.

As usual, Confidentiality goals involve complex proof strategies and complications with the theorem prover. This is evidence that when using the Multicast event model in comparison to the standard Unicast one. But we also can extract some new insights that come with these complications, since it requires better understanding of the protocol properties. Understanding these complications is required to guide the theorem prover through. In a worst case scenario it helps the understanding of the proof by not concealing the reasoning behind the automatic techniques of Isabelle/HOL.

Finally we can say that the increase in the effort for constructing the proof under a Multicast event model in comparison to a Unicast one is moderate, which justifies an implementation in a mixed environment using the Multicast primitives.

## 3.4 Conclusions

This chapter shows our main contribution for the extension of the inductive method. We showed how the method can be extended by a new message framework and how it can be used to represent all the other methods of message casting other than the Unicast available before.

We started with a review of the different categories of multicast focusing on their core properties that could be captured by our specification. We analysed the importance of unreliable multicast had over time in to the delivery of novel security goals. We also pointed that the establishment of Byzantine Agreement requires the reliability of the multicast model.

From that point on, we described our modification in the event theory to enable the reasoning regarding the Multicast event primitive and we showed our justification for the choices we did in our implementation. Then, we proved some of the basic lemmas that were derived from the Unicast implementation, so that we could make our new implementation usable. We then do an analysis of the re-interpretation of some well established concepts under the new Multicast framework.

Finally to corroborate our idea regarding the coverage of the Multicast implementation regarding the other message casting frameworks, we revisit the verification of a well known Unicast protocol under the new framework. We showed the specification and verification for it, trying to pinpoint the difficulty brought by the new framework. We then finalised by concluding that this newly introduced difficulty is small.

— *To know that one has a secret is to know half the secret itself.*
Henry Ward Beecher (1813 - 1887)

# 4 Secret Sharing Formalisation

Threshold security schemes are designed to provide resilience and controlled usage when we deal with a secret. Our secrets normally are cryptographic keys, or passwords that encrypt cryptographic keys, which must be controlled against malicious use or accidental loss. These secrets when shared by a threshold scheme achieve such properties. Under this scenario they can have a varied use such as providing privacy on electronic election protocols [110], providing resilience and controlled access in Public Key Infrastructures environments [83] and guaranteeing fair usage on dangerous weapons [6].

One way of implementing threshold schemes is by the use of secret sharing schemes. In a secret sharing scheme we have at first a secret and a set of trustees to whom the shares will be delegated. The original secret is not intended to be known by any of the trustees, and the scheme should not to allow collusion among them to acquire it. From this set of trustees we will derive our access structure based on our security policy and will create the number of shares we need to proceed with the scheme. This phase can include a trusted dealer or not depending on our needs. In the reconstruction phase, the shares are joined until we reach the threshold, and the secret can be recovered.

An access structure for a secret sharing scheme is by definition all subsets of the set of shares (trustees) that will trigger the reconstruction of the secret. Secret sharing schemes can be categorised depending on their access structure and the randomness included within the generated shares. In terms of introduced randomness within the shares, we have two different classifications coming from Information Theory analysis; we have unconditionally secure schemes, whose security is defined by having shares at least as long as the secret, increasing the amount of randomness introduced in each share. The second class, the not unconditionally secure schemes, have their shares not as long as the secret, trading security for efficiency. Also by extension they introduce less randomness, and they are not theoretically secure. They are still computationally secure, meaning that they are protected not by their theoretical construction but by the impossibility of computing all possibilities. Unconditionally secure schemes are normally considered in cases where share size does not play an important role in the context.

Threshold security schemes can be further classified in groups depending on the functionalities they yield. At first, we have those where we are interested only in the number of participating trustees, and a trigger for reconstruction, such as the classical Shamir's (§4.2.1) and Blakley's schemes (§4.2.2). Their main objective is to represent an access structure, as will be explained to account for controlled release and redundancy. No care is taken into how to manage the secret or the shares. In this scenario the only way of knowing you still hold the secret is by triggering a reconstruction. And the only way of redistributing shares is by re-sharing the secret with new parameters. These schemes introduce a series of limitations regarding the lifecycle of the secret. We cannot assert anything regarding shares compromised in previous runs since they can still trigger a valid reconstruction, nor to tell that we can reconstruct the secret without actually doing so.

Trying to address some of the problems explained above we see the introduction of verifiable secret sharing schemes by Chor ET al. [45]. They aim at guaranteeing that each trustee receive a valid share providing a verification token. But Chor's proposal suffers a drawback that is its interactiveness for the verification. Moreover, the complexity of the communication in the scheme is exponential over a series of nested broadcasts. Feldman [54] has proposed a non-interactive scheme for achieving verifiability on Shamir's scheme using a single broadcast for the verification function. Pedersen [102] later proposed another version of verifiability for Shamir's scheme also using a non-interactive verification scheme, but now using information-theoretic security instead of the hardness of the verification function.

Further extending the idea of verifiable secret sharing, Stadler [117] introduces publicly verifiable secret sharing schemes. The verification of the share's correctness regarding the secrets can be done by anyone, and not only the participants. In the scheme proposed by Schoenmakers [112], the shares corresponding to some secret S are encrypted by the dealer using the public key of the shareholders, which are then broadcasted. In a general setting, this brings the secret sharing setup very close to the proposition of public key cryptography, where the shares do not need to be revealed to anyone to verify the existence of the claimed secret.

Another major issue when using secret sharing schemes is the detection and identification of cheaters that are trying to subvert the recovery of a shared secret by supplying modified shares. Some schemes can detect cheating but not the party that cheated. This is generally done by a combination of one-way functions and broadcasts. Detecting the cheaters is harder, but some proposals exist to address this problem. In general the dealer can broadcast digests of the shares, and in this case the cheater can be detected using the public digests.

Finally some secret sharing schemes try to address the issue of changeability. Sometimes an explicit secret, such as a private key, must be re-shared according to a new access structure. This accounts for individuals leaving the set of trustees, or new trustees coming in, or a change of policy to the threshold for the secret. Changeability can be achieved by using proactivity, where we renew the shares without changing the secret. This concept was first defined by Herzberg ET al. [67]. The main advantage of proactivity is that even if the adversary obtains some shares he cannot obtain any useful information about the secret after the shares are renewed. The changeability of the threshold was discussed by Martin ET al. [81], where they used the intrinsic characteristics of plane geometry from Blakley's to get the first points for the shares and change it from planes to achieve a different threshold. Disenrollment of shares was discussed by Blakley ET al. [29] and by Martin [81]. The general form of disenrollment regards the combination of verifiability and changeability to enable the reconstruction of the secret and the re-sharing under the newly defined set of trustees.

Seeing this complex scenario of how the different secret sharing mechanisms operate and after a thorough research in their internal operation, we opted for an abstract modelling that would enable us to capture the most distinct characteristics of the different secret sharing schemes. We opted for implementing the abstract definition for secret sharing schemes in the inductive method using a similar approach as the one taken for public key cryptography. We divided the scheme into standard threshold secret sharing schemes, where our main goal was to represent the sharing and recovery of shares as done by Shamir and Blakley, and the publicly verifiable secret sharing schemes, where we introduced the existence of a public component that can be used to verify the possibility of reconstruction of the secret. We did not take into consideration problems arising from the identification of cheaters or the changeability of issue since they normally represent protocols themselves. This can be built upon our basic specification in the future if necessary.

Regarding the specification within the inductive method we had initially two options for the integration of secret sharing. The first and most complex one was the changing of the *msg* data type to account for a constructor *Share*. From this perspective we would have to build the infrastructure in a similar way as the one done for shared and public key cryptography. The second one, shown, accounts for a simpler integration that does not break backwards compatibility of the data type *msg* and that shields the shares with nonce properties. This approach uses ideas from Bella [20] who in his work divided the key space for his Kerberos specifications. This specification has some drawbacks regarding knowledge distribution, but proved to be sufficient to achieve confidentiality goals in the protocol scenarios we tested it with. Its specification and usage in Byzantine protocol are shown. Although only the simpler approach is shown in this thesis we did experimentations with the more complex one, but decided to drop it when our focus shifted from a Secret Sharing specification to the provision of different message casting frameworks.

This chapter started with the above summary regarding secret sharing and follows with a deeper view into access structures (§4.1). Access structures play a very important role in the way any threshold security scheme works and their understanding is paramount to construct the abstract definitions. We then will look into classical secret sharing schemes (§4.2) paying attention to the classical propositions from Shamir (§4.2.1) and Blakley (§4.2.2). They account for our abstract definition of secret sharing without verifiability. Similarly we will look deeper into verifiable secret sharing schemes (§4.3) and into the propositions from Feldman (§4.3.1), Pedersen (§4.3.2) and Stadler (§4.3.3). In our definition of secret sharing, verifiable secret sharing schemes will have a counterpart that enables the verification of the possibility of reconstruction. Finally we will show our specification of threshold cryptography that was used in the Franklin-Reiter sealed-bid auction protocol (§4.4), which is based on the shielding of the shares with nonce properties and the distribution on knowledge by the usage of *Notes* events.

## 4.1 Access Structures

In this thesis we only consider monotone access structures. These can be described by the property that if a group of shares is able to reconstruct a secret, so can a larger group. Monotonicity is important, since it controls the behaviour of the secret sharing scheme, regarding the collection of more than $t$ shares. It also establishes egalitarian importance to each share we create and distribute, enabling us to keep also a monotone threat model, which is already present in the inductive method.

Ito ET. al [73] defines that an access structure $\alpha$ must satisfy the following natural condition:

$$\forall B \in P(\{1, 2, ..., n\}))((\exists A \in \alpha)(A \subseteq B) \Rightarrow B \in \alpha)$$

where the set of groups $\alpha \subseteq P(\{1, 2, ..., n\})$, and where our users are labelled from $1, ..., n$, meaning that if a certain size group can recover a secret, a larger one also can. Later Benaloh and Leichter [28] named this natural access structure *monotone*, and also defined its counter property:

$$\forall B \in P(\{1, 2, ..., n\}))((\exists A \in \overline{\alpha})(B \subseteq A) \Rightarrow B \in \overline{\alpha})$$

meaning that if a certain sized group cannot recover a secret, neither can an even smaller group.

Although we see the importance of *non-monotone* access structures in threshold cryptography mechanisms, specifically to represent veto capabilities in protocols, we decided not to cover them in our models and by extension in our specification efforts. The introduction of *non-monotone* access structures imply the modification of the threat model used by the inductive method (see Section 2.2.2). These modifications would break compatibility with actual proofs, since we would have to start establishing importance to knowledge acquired by the Spy and specialise behaviour for that. This also would augment the capabilities of the Dolev-Yao [51] attacker, giving him capacities for creating Denial-of-Service attacks, which is not achievable in a straightforward manner within the actual inductive method. Non-monotonic access structures would imply a full redesign of the threat model, which is way beyond the scope of the thesis.

Concerning monotone $\alpha$ access structures authorisation groups, Iftene [70] defines them as:

**Definition 33.** $\alpha = \alpha_{min} \cup cl(\alpha_{min})$

$\alpha_{min}$ are the minimal authorised groups in $\alpha$ access structures, which will represent the minimal number of participants that can enable a reconstruction. It is shown in Definition 34

**Definition 34.** $\alpha_{min} = \{A \in \alpha \mid \forall B \in \alpha \setminus \{A\})(\neg(B \subseteq A))\}$

The $cl(\alpha_{min})$ to any $\alpha_{min} \subseteq P(\{1, 2, ..., n\})$ is defined as:

**Definition 35.** $cl(\alpha_{min}) = \{A \in P(\{1, 2, ..., n\})|(\exists C \in \alpha_{min})(C \subseteq A)$

The unauthorised access structure $\overline{\alpha}$ is then specified by the set of the maximal unauthorised group:

**Definition 36.** $\overline{\alpha}_{max} = \{A \in \overline{\alpha} | (\forall B \in \overline{\alpha} \setminus \{A\})(\neg(A \subseteq B))\}$

The above definitions help us to precisely picture the peers that can participate in the secret sharing reconstruction and those who cannot. The $\alpha$ set defines all the possible authorised subsets that will trigger a reconstruction. $\alpha_{min}$ represents the threshold that must be achieved for the reconstruction to take place, and $cl(\alpha_{min})$ represents the closure between $\alpha_{min}$ and $\alpha$. To illustrate we present an example. It also shows that $\alpha_{min} \neq \emptyset$, since any group would be able to recover the secret. These structures are known as Sperner Systems, and are out of our intended scope.

Lets consider our $n = 4$ and the $\alpha$-access structure as $\alpha = \{\{1,2\},\{1,2,3\},\{1,2,4\},$ $\{1,2,3,4\},\{3,4\},\{1,3,4\},\{2,3,4\}\}$. From this $n$, $\alpha$ and the definitions we obtain that $\alpha_{min} = \{\{1,2\},\{3,4\}\}$, $\overline{\alpha} = \{\{1\},\{2\},\{3\},\{4\},\{1,3\},\{1,4\},\{2,3\},\{2,4\}\}$ and $\overline{\alpha}_{max}$ $= \{\{1,3\},\{1,4\},\{2,3\},\{2,4\}\}$.

The example shows another implicit property called *connection*. The $\alpha$-access structure is said to be *connected*, when any user $i$ belongs to some minimal authorised group [36]. Other properties regarding a $\alpha$-access structure is its *rank* and *co-rank*, which are defined as the maximum and minimum number of participants in a minimal authorised group. If both are equal to some positive integer $r$ we say that $\alpha$ is *r-homogeneous*, and every minimal authorised group has exactly $r$ members. The intersection number of $\alpha$ is the maximum number of participants that will be in each two different minimal authorised groups.

Understanding access structures is paramount for understanding any secret sharing construction as well as their internal operation. With these basic concepts regarding access structure laid down, we will look to implementations of access structures as secret sharing mechanisms starting with the plain propositions of secret sharing scheme made independently by Shamir and Blakley. We will see the evolution of such schemes into verifiable ones and will conclude with our Isabelle/HOL specification of threshold cryptography for the inductive method.

## 4.2 Threshold Secret Sharing Schemes

Threshold secret sharing schemes are those schemes where all participants have equal importance and only the number of participants in the reconstruction phase is important. In these schemes we divide our secret in $n$ parts and establish a threshold $k$ in our access structure scheme that will re-enable the reconstruction in the presence of $k$ shares. We will define the access structure for $(k, n)$-threshold secret sharing schemes, as follows:

**Definition 37.** *Let $n \geq 2$ and $2 \leq k \leq n$. The $\alpha$-access structure is defined by*

$$\alpha = \{A \in P(\{1, 2, ..., n\}) \mid |A| \geq k\}$$

We can easily obtain the other important properties defined so far (Definitions 34, 35, 36). We have $\alpha_{min} = \{A \in P(\{1,2,...,n\}) \mid |A| = k\}$, $\overline{\alpha}_{max} = \{A \in P(\{1,2,...,n\}) \mid$ $|A| = k - 1\}$, and $\overline{\alpha} = \{A \in P(\{1,2,...,n\}) \mid |A| \leq k - 1\}$.

From Definition 37, we can draw the next example:

If $n = 4$ and $k = 3$, the $\alpha$-access structure following Definition 37 is
$\alpha = \{\{1,2,3\},\{2,3,4\},\{1,2,4\},\{1,3,4\},\{1,2,3,4\}\}$,

$\alpha_{min} = \{\{1,2,3\},\{2,3,4\},\{1,2,4\},\{1,3,4\}\}$,
$\overline{\alpha} = \{\{1\},\{2\},\{3\},\{4\},\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{3,2\},\{2,4\},\{3,4\}\}$ and
$\overline{\alpha}_{max} = \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{3,2\},\{2,4\},\{3,4\}\}$.

The properties derived in the above example are constructed distributing the $n$ elements in $P$ permutations with $P \geq k$ size ($\alpha$), $P = k$ size ($\alpha_{min}$), $P \leq k$ size ($\overline{\alpha}$) and $P = (k-1)$ size ($\overline{\alpha}_{max}$).

Following this section, we will cover details on how different propositions cover the above stated properties regarding threshold secret sharing schemes. We will cover the classic methods independently created by Shamir (4.2.1) and Blakley (4.2.2) in 1978. In Blakley's approach our focus is to cover his view regarding the different attacks a threshold scheme is subject to.

### 4.2.1 Shamir's Scheme

Shamir's scheme [114] has its mathematical base on a polynomial interpolation, where given any $k$ pairs $(x_1, y_1), ..., (x_k, y_k)$, where $x_i \neq y_i$, then for all $1 \leq i < j \leq k$, $\exists! P(x)$ of degree $k-1$, such that $P(x_i) = y_i$ for all $1 \leq i \leq k$.

From that, the secret becomes the free coefficient of a random polynomial with degree $k-1$. This operation happens over the field of positive integers modulo large primes. The shares $S_1, ...S_n$ are generated from $S_i = P(x_i)$, where $x_i, ...x_n$ are distinct values. When we have the set of shares $\{S_i | i \in A\}$ , and where $|A| = k$, the secret is easily obtained by Lagrange's polynomial interpolation formula.

$$S = \sum_{i \in A} (S_i \cdot \prod_{j \in A \setminus \{i\}} \frac{x_j}{x_j - x_i})$$

When we have $|A| > k$ shares, we can simply discard randomly any share that makes $|A|$ bigger than $k$, making reconstruction just from $k$ shares. This is important: we should not build proofs on who actually participated in the reconstruction, since that, with Shamir's scheme they are just discarded randomly, and cannot have participation guaranteed just by the scheme.

Another important issue is about correctness of the secret, since the scheme cannot detect any attack coming from one of the trustees giving back the wrong share, leading to the reconstruction of the wrong value. We cannot detect who gave us the wrong share, since the dealer does not keep them.

Shamir also proposed binding $x_i = i$, $\forall 1 \leq i \leq n$, making in this case the secret reconstruction simplified to

$$S = \sum_{i \in A} (S_i \cdot \prod_{j \in A \setminus \{i\}} \frac{j}{j - i})$$

for any set of shares $\{S_i | i \in A\}$ , and where $|A| = k$. He also highlighted that his scheme has some very interesting features. The size of the share is the same size of the secret, making it theoretically secure. Shares can be added or excluded without affecting the other shares, making such changes to the shares without changing the secret. This feature will come to light later when we will discuss proactive secret sharing schemes and how well Shamir's proposal is adaptable to this approach.

### 4.2.2 Blakley's Scheme

Blakley scheme [30] makes the secret an element of a vector space $GF_q^k$, and the shares are constructed as $n$ distinct $(k-1)$-dimensional hyperplanes that intersect at the secret element. A $(k-1)$-dimensional hyperplane can be defined as a set of the following form:

$$\{(x_1, ..., x_k) \in GF_q^k | \alpha_1 \cdot x_1 + ... + \alpha_k \cdot x_k = \beta\},$$

where $\alpha_1, ..., \alpha_k$,$\beta$ are random elements of the field $GF_q$.

Although Blakley's scheme is not perfect because any group of two or more share holders together can know that the secret $S$ is at the intersection of their shares (planes), Blakley's work [30] is important to our formalisation efforts. He defines very concisely a threat model into which a normal threshold secret sharing scheme is inserted.

He starts by asking: " What must the secret be guarded against?", and comes with at least three types of incident [30]:

- **An abnegation incident :** the access to the information is lost by losing access to shares. We have less shares than $k$. Abnegation incidents can be sub-classified in other three smaller groups:
  - Destruction: the share is destroyed and its recovery is impossible, having as an example the sudden death of a security officer holding it.
  - Degradation: the security officer forgets how to access his share, and in the event of request will produce any random data just to participate on the reconstruction. No bad intention is regarded in the action.
  - Deflection: the security officer is compromised by the attacker and passes him information about the share, neglecting to tell the organisation that his share was compromised.
- **A betrayal incident:** the shares are in possession of the attacker, giving him assurance of the share contents and precedence. This type of incident has also two sub-classifications:
  - Deflection: as shown before, but this time the leakage is complete in terms of the share information.
  - Dereliction: the security officer passes his share to the attacker, but collects it back to avoid detection from the organisation. An example could be a share restricted to a smart card.
- **A combination incident:** is an abnegation incident that is also a betrayal incident, being the main type a defection.

Even though Blakley brings to light a threat model to secret sharing in his early paper, he also states that his model does not take into account any malfeasance, misfeasance or even nonfeasance from the internal security officer. This feature is present in the threat model because it is a pre Dolev-Yao [51] paper, where the security community was not fully aware of the role internal attacks play in any reasonable security policy.

He also considers how to recognise the right key when it comes out from the secret sharing mechanism. He points out that when an attacker collects more than $k$ shares and each of the $k$-sized sets solves to the same secret, the attacker has assurance that he has collected the right shares and the secret protected by them. Taking his example [30], if we have $n = 9$ and $k = 4$, any set of $k + 1$ shares is enough to an attacker to have certainty the secret was recovered, since we will be able to reconstruct the same share 5 times. Taking the attacks described by Blakley, we will need $k + 2$ to detect a betrayal attack.

## 4.3 Verifiable Threshold Secret Sharing Schemes

One of the main problems of the standard threshold secret sharing schemes is that we must consider the participating peers honest. To be able to operate with the threshold properties even in the presence of a malicious dealer we must use a verifiable threshold secret sharing scheme. A threshold secret sharing scheme can be called verifiable if it includes some extra information that allows the peers involved in keeping the shares to verify they keep a consistent secret.

The verifiability problem was first discussed by Chor ET al. [45], who introduced the notion of verifiable secret sharing schemes where every user can verify that he or she has a valid share for a specific run. The proposition from Chor ET al. is interactive and makes use of nested broadcasts, which makes the communication exponentially complex. Feldman [54] and later Pedersen [102] proposed new non-interactive verifiable schemes that we will see in depth. The problem of cheating in the reconstruction phase by giving to the dealer a deliberately corrupted share has been discussed by McEliece [85] and Tompa [123]. Verifiable secret sharing schemes are also a solution for this problem, since the shares presented during the reconstruction phase may be verified with respect to the distribution phase.

We will cover in depth the proposition from Feldman for a verifiable scheme, followed by the Pedersen proposal. Later we will cover Stadler's proposition, augmented by Schoenmakers', for publicly verifiable secret sharing over which we based the second part of our specification for threshold cryptography.

### 4.3.1 Feldman's scheme

Trying to avoid the disadvantage concerning the interactiveness from Chor ET al., Feldman [54] proposed a non-interactive scheme for achieving verifiability in Shamir's proposition. The new scheme's main idea is to use a one-way function $f$ such that $f(x + y) = f(x) \cdot f(y)$ and to broadcast $f(a_0), ..., f(a_{k-1})$, where $P(x) = a_0 + a_1x + ... + a_{k-1}x^{k-1}$ is the polynomial for Shamir's scheme.

Feldman's scheme starts with the generation of two primes, namely $p$ and $q$, such that $q|(p-1)$ and $\alpha \in Z_p^*$ is an element of order $q$. These numbers are made public. Following that, the dealer generates a polynomial $P(x) = a_0 + a_1x + ... + a_{k-1}x^{k-1}$ over $Z_q$ such that $a_0 = S$ and makes public $\alpha_i = \alpha^{a_i} \bmod p$ for all $0 \leq i \leq k-1$ shares. The dealer will then securely distribute the shares $I_i = P(i)$ for all the trustees belonging to the access structure. Each user then can verify the correctness of the received share $I_i$ by testing the following condition:

$$\alpha^{I_i} \bmod p \overset{?}{=} \prod_{j=0}^{k-1} \alpha_j^{ij} \bmod p$$

The importance of understanding Feldman's proposition to our abstract definition of threshold cryptography in the inductive method regards the way it enables the verification of the validity of shares by distributing extra information in a similar way public key cryptography does. By doing it non-interactively, this gives us the insight on modelling two different types of primitives, which later we extend to address publicly verifiable secret sharing.

### 4.3.2 Pedersen's scheme

One of the disadvantages of Feldman's scheme is that the application of function $f$ over $S$ is broadcasted, what makes the security of the scheme dependent on the hardness of the inverting function $f$. Pedersen [102] proposed a different non-interactive and information-theoretic secure verifiable variant of Shamir's scheme.

Pedersen' s scheme starts generating the primes $p$ and $q$ such that $q|(p-1)$, and $g, h \in Z_p^*$ which are elements of order $q$. These numbers are made publicly available. The dealer then broadcasts $E_0 = g^S h^t \bmod p$, where $t \in Z_q$. After that, the dealer generates $P(x) = S + P_1 x + ... + P_{k-1} x^{k-1}$ and $Q(x) = t + Q_1 x + ... + Q_{k-1} x^{k-1}$ over $Z_q$. He will then make public $E_i = g^{P_i} h^{Q_i} \bmod p$ for all $1 \leq i \leq k-1$. Finally he securely distributes the share $I_i = (P(i), Q(i))$ belonging to the $i$-trustee for all $1 \leq i \leq n$. Each trustee then can verify the correctness of the received share $I_i$ by testing the following condition:

$$g^{S_i} h^{t_i} \bmod p \overset{?}{=} \prod_{j=0}^{k-1} E_j^{ij} \bmod p$$

The importance of understanding Pedersen's scheme regards the introduction of information theoretic security. Our specification considers that sharing scheme as perfect, like the other crypto specification within the inductive method specification. This point should be called to attention since in the case of secret sharing schemes, we have indeed functions that are not based on computational complexity, thus giving more fidelity to our specification if compared to the standard cryptography one.

### 4.3.3 Publicly Verifiable Secret Sharing Schemes

Stadler's [117] scheme introduces the idea that the correctness of the shares with respect to the secret should be verifiable by everyone and not only by the participants. This approach is called a publicly verifiable secret sharing scheme. The shares are encrypted by the dealer for the secret $S$ using the public keys of the trustees. The encrypted shares $ES_i = e_{k_i}(I_i)$ for $1 \leq i \leq n$ are broadcasted and made available to anyone. A public function is provided for testing the validity of the encrypted shares. If the broadcasted shares pass the test, then their decryption will lead to the correct reconstruction of the secret if the dealer was honest.

The verifiable secret sharing scheme as proposed by Stadler and perfected by Schoenmakers [112] is composed of three phases: setup, distribution and reconstruction phase.

During the setup phase a group $G$ of prime order $q$ and some generators $\alpha$ and $\beta$ of $G$ are generated and broadcasted by the dealer. Each of the trustees generates a secret key $t_i \in Z_q^*$ and broadcasts $y_i = \beta^{t_i}$ as its public key.

Starting the distribution phase, the dealer generates the polynomial $P(x) = a_0 + a_1x + ... + a_{k-1}x^{k-1}$ over $Z_q^*$ such that $a_0 = s$ for some $s \in Z_q^*$ and makes public $\alpha_i = \alpha^{a_i}$ for all $0 \leq i \leq k-1$. The secret will be $S = \beta^s$. The dealer will broadcast the encrypted shares $Y_i = y_i^{P(i)}$ for all $1 \leq i \leq n$. To demonstrate correctness of the encrypted shares the dealer shows a proof of knowledge of the value $P(i)$ such that $X_i = \alpha^{P(i)}$ and $Y_i = y_i^{P(i)}$, where $X_i = \prod_{j=0}^{k-1} \alpha_j^{ij}$ for all $1 \leq i \leq n$.

The reconstruction phase requires each trustee to use its own private key to transform the encrypted share $I_i = \beta^{P(i)}$ by computing $I_i = Y_i^{t_i^{-1} mod q}$ and to forward the value to the dealer. A group of at least $k$ trustees can recover the secret $S$ by the following function:

$$S = \prod_{i \in A} I_i^{\prod_{j \in A\ \{i\}} \frac{j}{j-i}}$$

Publicly verifiable secret sharing uses zero knowledge proofs to achieve an equivalent security to public key cryptography while also adding the threshold mechanisms. The importance of understanding publicly verifiable secret sharing schemes in our effort to abstractly define threshold cryptography within the inductive method is that it enabled us to refine our specification further regarding public shares. In our experimentations with the full embedding of the secret sharing primitives in the inductive method, the assumption of existence of such methods enables us to distribute the knowledge of public shares in a very similar way as the method treats public cryptographic keys.

Having done a thorough review on secret sharing and its major properties we will be showing in the next section our most simple specification that was used to verify confidentiality in the Franklin-Reiter sealed-bid auction protocol. Our main goals in this experiment were to produce a usable secret sharing abstract definition while not changing the data type *msg*, and keeping backwards compatibility. We also did some experimentation dividing the space for nonces.

## 4.4 Formalisation of Threshold Cryptography

Following the approach stated above, we decided for a specification of the threshold cryptography in a simpler fashion for use with the sealed-bid auction protocol we verify in Chapter 5. We opted for shielding all the threshold cryptography primitives as Nonces when in transit between peers. Once peers are able to reconstruct the shares, we give them the values recovered by the reconstruction process by the usage of a *Notes* event. We opted for this specification instead of the full one based on changes in the *msg* data type because our main focus shifted towards the verification of Multicast based protocols instead of threshold cryptography. The specification of the threshold cryptography specification is present in our own version of *Public.thy* called *PublicSS.thy*

Our specification for threshold cryptography starts with the specification of shares as a specific class of non guessable numbers (nonces). By doing so we need to be able to differentiate nonces into different classes, so that they have a different space in the infinite set of possible nonces for shares and for nonces. Due to intrinsic issues regarding the specification of the Franklin-Reiter Sealed-Bid Auction protocol, we are introducing a new division in the nonce space so that we can differentiate nonces as freshness components as well as shares and session identification components. In the sealed bid auction protocol, the freshness of a bid is a function between the freshness of the casting itself and the freshness of the specific session. Our division of the nonce space is done with the declaration from Definition 38.

**Definition 38.** *Secret sharing declaration*

```
consts
sessionIDs :: "nat set"
shares :: "nat set"
```

Definition 38 simply creates two new constants typed to the set of natural numbers so as nonces are in the inductive method specification. To be able to specify the disjoint properties we need to create the different spaces in the already existing nonce space we specify *sessionIDs* as shown in Definition 39.

**Definition 39.** *Specification of sessionIDs function*

```
specification(sessionIDs)
sessionIDs_disj1 [simp]: "sessionIDs ∩ shares = {}"
```

Specifying the total disjunction for *sessionIDs* and *shares* is done by stating their intersection is empty. With Definition 39 we have the possibility of having the division of the nonce space into three parts. We have those nonces belonging to *sessionIDs*, like $Nonce\ X \in sessionIDs$, those belonging to shares, like $Nonce\ X \in shares$ and those not belonging to either, like $Nonce\ X \notin sessionIDs \wedge Nonce\ X \notin shares$. This will help us address a specific problem we face in the FR-Auction protocol, where session identification must be repeated for the different bids.

With the disjoint classes for nonces specified, our next step is the definition of the function for representing the threshold cryptographic primitives. We need to define the important characteristics of threshold cryptography primitives. We opted for the specification of two different types of threshold schemes, trying to capture in the most abstract way their functionality. As our discussion showed before, having a specification that enables us to represent all types of threshold cryptography is almost infeasible. We opted then to implement a generic monotonic scheme that is capable of capturing the standard schemes based on the initial ideas of Blakley and Shamir, and a secondary scheme that is capable of abstracting the publicly verifiable secret sharing schemes, giving them a publicly verifiable counterpart.

To achieve such specification we defined three functions, as shown in Definition 40. We defined a function called *share* to represent the first class of primitives discussed above. It takes as argument a reconstruction threshold, an agent list to which the shares will be distributed, the agent to which this share belongs and the message we want to share with the threshold mechanism. It returns a natural number that will be formalised as a nonce. We also define a function to represent the second class of primitives, *priv_share*. The function *priv_share* takes as arguments the same arguments as *share*: a reconstruction threshold, an agent list to which the shares will be distributed, the agent to which this share belongs to and the message we want to share into the threshold mechanism. It also returns a natural number. We must note that this specification do model the information-theoretically secure schemes precisely, since the share can be any natural number. It is not constrained to be bigger or equal to the size of the secret. We also specified a function *invShare* enabling us to relate the public counterpart of a share to its private one. It takes a share as input and returns the counterpart for that share.

**Definition 40.** *Secret sharing primitives declaration*

**consts**
```
share ::  "[nat => (agent list) => agent ] => msg => nat"
priv_share ::  "[nat x (agent list) x agent ] => msg => nat"
invShare ::  "nat => nat"
```

The specification for the functions declared at Definition 40 is done separately. To define the initial properties for the functions *share* and *priv_share* we opted for an axiomatic assertion, as shown in Definition 41. Both functions have similar properties and we defined them on the relation between the sharing parameters and the message being taken. So equal shares will have the same parameters as well as will be sharing the same payload. The same is true for *priv_share*.

**Definition 41.** *axioms*

**axioms**
```
share_def [dest!]:  "share p X = share q Y ==> (X = Y) ∧ (p = q)"
priv_share_def [dest!]  :  "priv_share p X = priv_share q Y ==>
                                     (X = Y) ∧ (p = q)"
```

This approach is a set of simplifications from other experiments we carried. It enables us to capture the essence of the threshold mechanisms. The imprecision concerns the representation of some sharing schemes that if executed twice under the same parameters will return different values for the shares. In our case to differentiate shares we take the parameter and the payload. We cannot address this specificity some methods have. This will make it extremely difficult for us to represent the re-issuing of shares in a related manner.

The function used to relate the counterparts in the case of verifiable schemes is specified as shown in Definition 42. The inversion of shares to their related counterparts is done by specifying that the double application of the function *invShare* will yield back the value given to the function initially. This idea was borrowed from the initial specification of public key cryptography done within the inductive method.

**Definition 42.** *Specification of InvShare function*

**specification**`(invShare)`
```
invShare [simp]:  "invShare (invShare S) = S"
```

Having the private counterpart of the verifiable threshold scheme representation available together with a function that is able to invert it, we can define its public counterpart. The relation between the private and the public shares counterparts will by done by half-step of the *invShare* function as shown by the abbreviation on Definition 43.

**Definition 43.** *pub_share definitions*

**abbreviation**
```
pub_share::  "[nat x (agent list) x agent ] => msg => nat" where
"pub_share p X == invShare (priv_share p X)"
```

In this way we again simplify our specification stating that this relation is one-to-one, as it is now always the case for the various methods.

A next step is to create the differentiation of the verifiable threshold schemes from the non-verifiable ones, as well as creating the differentiation of public and private shares. This is done by an axiomatic assertion as shown by Definition 44

**Definition 44.** *axioms*

**axioms**
```
pub_share_non_priv_share [iff]:  "pub_share p X ≠ priv_share p X"
share_non_priv_share [iff]:  "share p X ≠ priv_share p' X'"
share_non_pub_share [iff]:  "share p X ≠ pub_share p' X'"
```

The first axiom of Definition 44 states that a public share is different from its private counterpart. By stating this relation we guarantee that although related by the function *invShare* a public share cannot collide with its private counterpart. The other two axioms regard the differentiation of verifiable and non-verifiable threshold mechanisms. The first states that no private share is equal to a share, and the second states that no public share is equal to a share.

Our final step in the specification of the threshold cryptography support we will need for the verification of the Byzantine protocol in the next chapter is to determine the membership of the three different share types to the *shares* set defined above. This is done as shown by Definition 45.

**Definition 45.** *Specification of shares functions*

**specification**`(shares)`
```
shares_share [iff]:  "share p X ∈ shares "
shares_priv_share [iff]:  "priv_share p X ∈ shares "
shares_pub_share [iff]:  "pub_share p X ∈ shares "
```

A non-verifiable share is in *shares*, the private part of a verifiable share is in *shares* and the public part of a verifiable share is also in *shares*. The set *shares* represents all the shares available and is by definition disjoint from the session identification nonces as done by Definition 39.

Having specified the basic support for the usage of threshold cryptography in the inductive method, we now need to prove some lemmas to give to Isabelle/HOL's classical reasoner and simplifier the knowledge to treat the specification according to what we need. The next subsection (§4.4.1) will show us such basic lemmas.

### 4.4.1 Basic Lemmas

The lemmas are the basic knowledge Isabelle/HOL requires to reason about the specification we made in the previous section. Our idea here is to present them briefly so that we can understand the reasoning behind what Isabelle/HOL will do with its automation procedure.

We start the lemmas that allow Isabelle/HOL to infer the inequalities we axiomatically made in Definition 44, and they are shown in Lemma 39. The idea of this set of lemmas is to give Isabelle/HOL knowledge to quickly infer all the variations from Definition 44. To do that, we directly state all the permutations and prove them using the axioms.

**Lemma 39.** *Secret sharing abstract definition lemmas*

```
lemma [iff]:  "priv_share p' X' ≠ share p X"
lemma [iff]:  "priv_share p' X ≠ share p X"
lemma [iff]:  "priv_share p X ≠ share p X"
lemma [iff]:  "pub_share p' X' ≠ share p X"
lemma [iff]:  "pub_share p' X ≠ share p X"
lemma [iff]:  "pub_share p X ≠ share p X"
lemma [iff]:  "priv_share p X ≠ pub_share p X"
```

Our next step is to prove lemmas regarding the specification made in Definition 39 for *sessionIDs* and *shares*. The disjoint specification over the range of possible nonces is enough to prove Lemmas 40 and Lemma 40.

On Lemma 40 we give Isabelle/HOL the destruction rule that states the implication that if $X$ is in *sessionIDs* then it is not in *shares*. This lemma is proven by tactic auto.

**Lemma 40.** *sessionIDs_not_shares*

```
sessionIDs_not_shares [dest]:  "X ∈ sessionIDs ==> X ∉ shares"
```

Conversely on Lemma 40 we prove the other side of the disjoint set, so that we can give to Isabelle/HOL another destruction rule which states that if $X$ is in *shares* it is not in *sessionIDs*. Its proof is also done by tactic auto.

**Lemma 41.** *shares_not_sessionIDs*

```
shares_not_sessionIDs [dest]:  "X ∈ shares ==> X ∉ sessionIDs"
```

Another important lemma regarding the disjoint specification is stated by Lemma 42. As the space for *sessionIDs* is disjoint to the space for *shares*, if $X$ is in one side and $Y$ in on the other side we can conclude the are not equal.

**Lemma 42.** *not_in_shares_not_equal*

```
not_in_shares_not_equal [simp]:  "X ∈ shares ∧ Y ∈ sessionIDs ==>
                                  X ≠ Y "
```

Another set of simplification rules was proven so that we can make use of the specification from Definition 45. By being shares, a share, a private share and a public share do not belong to *sessionIDs*. This is stated in each one of the three cases in Lemma 43, Lemma 44 and Lemma 45.

**Lemma 43.** *share_not_sessionIDs*

```
share_not_sessionIDs [simp]:  "share p X ∉ sessionIDs"
```

**Lemma 44.** *priv_share_not_sessionIDs*

```
priv_share_not_sessionIDs [simp]:  "priv_share p X ∉ sessionIDs"
```

**Lemma 45.** *pub_share_not_sessionIDs*

```
pub_share_not_sessionIDs [simp]:  "pub_share p X ∉ sessionIDs"
```

The Lemmas (43, 44 and 45) are proven bay appealing to Lemma 40 and to the specification of share shown on Definition 45.

With this we concluded the specification of our threshold cryptography support. This was intended to minimally specify threshold cryptography under the inductive method; to properly claim support to it we need its full embedding within the data type msg. We conducted this experimentation to detect how easy would it be to extend the method without breaking backward compatibility, and it seemed to be reasonably easy. We focused on not addressing too much detail regarding these new cryptographic primitives what made it easier.

We also must stress that with the shifted focus toward the specification of different message casting support, we did not include in this specification the support for reconstruction, since we used it only for proving confidentiality of the shares. The account for threshold reconstruction was partially specified in our experimentations regarding the changing of the data type *msg*.

### 4.4.2 Summary

We started with a thorough review of threshold cryptography mechanisms with a special focus to secret sharing schemes. We defined the concept of access structures and narrowed our scope to monotonic threshold mechanisms. We looked deeper into the classical mechanisms introduced by Shamir and Blakley, trying to base our decision into specifying threshold schemes in a parallel fashion to the standard cryptography already supported by the inductive method. We also covered the different methods developed for verifiable secret sharing, giving special focus to Feldman, Pedersen and Stadler. With this review of the secret sharing constructions, we established the basis for abstractly specified secret sharing support within the inductive method.

We then showed our specification in Isabelle/HOL for extending the inductive method to support threshold cryptography. We opted for a simpler approach in which we divided the nonce space for using nonces to shield the shares so that we could represent them in transit between the peers. The reconstruction of such shares is given to the peers through the usage of Notes events for the secret. We conclude our specification description by showing some basic lemmas we gave to Isabelle/HOL to enable it to reason about our definitions.

Summarising, in this chapter we showed a small contribution we had during the verification of a Byzantine security protocol based on multicast that required the support for threshold cryptography. The threshold cryptography specification shown here is an experiment we conducted with the sole purpose of proving confidentiality of the shares for the Franklin-Reiter sealed-bid auction protocols.

— *Distrust and caution are the parents of security.*
Benjamin Franklin (1706-1790)

# 5 Verifying a Byzantine Security Protocol Based on Multicast

Byzantine Agreement Protocols, or Byzantine Security Protocols, are designed to represent security a property in which achieving consensus is an important factor to the achievement of the protocol's security goals. Such properties concern either the ability to resist corruption or loss of some information by peers in the protocol which provides availability, or, to enable controlled release of information, where a consensus must be reached prior to disclosure.

Such Byzantine Security Protocols are able to detect which nodes have been compromised or failed, with a series of synchronised, secure rounds of message exchanges. These message exchanges are generally done using more complex communication frameworks than unicast, such as the different types of multicast. Also the provision of such consensus properties are not trivial using standard cryptography, thus Byzantine Security Protocols demand the introduction of novel cryptographic primitives, such as threshold based cryptography schemes.

Our efforts shown in the previous chapters for extending the inductive method were targeted to the verification of secrecy in a Byzantine Security Protocol. To be able to properly represent protocols based on non-unicast communication and being capable of supporting non-standard cryptographic primitives, we extended the inductive method as shown in Chapters 3 and 4 . Although our focus is in the provision of a novel specification for message exchanging in the inductive method, clearly our motivation comes from the inability of verifying Byzantine Security Protocols with the inductive method prior to our extensions.

This chapter makes use of the previous contributions to put in practice the verification the Franklin-Reiter Sealed-Bid Auction Protocol [58]. It is based in the combination of multicast and unicast, and makes use of non-standard cryptographic primitives based on threshold cryptography.

This protocol was conceived to enable bidders to cast their bids, distributing them among a pool of servers running the desired auction. Following the usual sealed-bid approach, all bids are sent encrypted, and in this case divided so that no single server can recover them. Once the bidding period closes, all correctly running servers will agree and start the next stage of the protocol. They will then agree on the winner and will check the availability of funds for the bidder to pay for the item. This is done using an off-line electronic cash scheme. Once each server independently agreed the winner, they will send the winning bidder a token for him to collect the item. Most of the cryptography used in the system is novel and based on threshold schemes. The sealed-bid auction protocol can be extended to provide anonymous bidding.

The setup of this protocol is Byzantine-secure because all the threshold mechanisms used in the sharing of the bid, and in the achievement of consensus for the closure and verification of the bids, can cope with partial corruption of peers. This protocol also introduces an unusual construction: a mixture of different message casting frameworks. This feature enables the check of our message casting specification, especially the correct distribution of knowledge within different modes of operation.

This chapter is structured with this initial introduction followed by a deeper view of the Franklin-Reiter sealed-bid auction protocol (§5.1). We describe the protocol by taking a look in the threat model as well as the properties the authors claim they achieve (§5.1.1). This section also contains three descriptions of the multicast assumptions taken by the authors (§5.1.2), the electronic money abstraction tailored to the protocol (§5.1.3), and the definition for a novel cryptographic primitive called verifiable signature sharing (§5.1.4), as well as, the main protocol description (§5.1.5). The protocol description subsection will show the messages we formalise later on, and it is followed by the description of the extension in the sealed-bid auction protocol to enable anonymity (§5.1.6). This section will end with some known issues we had already before the analysis (§5.1.7). Some of these issues were pointed out by authors, other from observations of our own.

The second part of the chapter (§5.2) comprises the modelling and verification we carried out using our specification of the sealed-bid auction protocol described in the first part. We will first present the specification of some auxiliary functions we developed to help us simplifying the reasoning (§5.2.1), which will be followed by the inductive specification of the protocol (§5.2.2). We then start the verification process (§5.3), where we first show the proofs constructed for the auxiliary function (§5.3.1) and follow on showing the verification of some key properties of the protocol to exemplify the usage of our multicast events theory (§5.3.3). Although we did not conclude the verification of the sealed bid auction protocol, the process of specifying it and constructing proofs expose some of its weaknesses (§5.3.4). We conclude the chapter with considerations regarding this verification exercise (§5.4).

## 5.1 The Franklin-Reiter Sealed-Bid Auction Protocol

Franklin and Reiter [58] proposed in 1996 a protocol to enable the construction of a distributed trusted service capable of executing sealed-bid auctions by using threshold cryptography primitives and extended multicast properties. Their effort was based on the view that some financial vehicles needed special requirements to be adequately implemented electronically and these were not properly addressed until that moment.

They focused their study on sealed-bid auctions due to their importance in conducting business and buying processes for governments and organisations. They were also motivated by the intrinsic novel security requirements. Among these we can cite the requirements for fairness in the timing of closure and how winners are identified. These protocols are also susceptible to attacks by corrupted internals leaking information during the bid collection phase and needed to be protected by a threshold mechanism for consensus.

Their objective was to provide a sealed-bid auction service that is guaranteed to declare a winner, and also to collect payment from only that bidder, while guaranteeing that no bid was revealed before the agreed bid opening time. Moreover, that the system should be resilient to malfeasance of any auction house insider.

### 5.1.1 Threat Model and Claimed Properties

Sealed-bid auctions are composed of at least two different phases: a phase where bids are made and sealed for future release, and another phase where the bids are revealed and the winner is declared. During the first phase, an arbitrary number of bidders interested in the item being bid upon can submit arbitrary many sealed bids to the auction. Once the bidding phase is finished, we start the second phase where the bids are opened. The winner is determined following the bidding rules, and the winner is announced. The determination of the winner is done by rules pre-established for the auction taking place, which generally is deterministic.

Sealed-bid auctions introduce a series of novel security properties that are brought to the information security scenario by Franklin and Reiter's protocol. Fairness requires the secrecy of the bids prior to the closing of the bidding period. This makes the proposed protocol very time oriented, since the time of disclosure of bids is crucial for determining the winner. A second property for sealed-bid auctions is the non repudiation of the bids, ensuring that the money promised in the bidding phase can be collected from the auction winner. Also due to the secrecy requirements for the bids, while in the bid casting phase, it is usually difficult to give the bidder confidence in the auction process without revealing the bids, so a distributed trust system is paramount for any computational solution for the problem.

The threat model for sealed-bid auctions is varied but certainly starts with the trust in the auction house and inside peers. Any reasonable threat model to a sealed-bid auction protocol should take into account attacks from insiders, collusion among peers up to a certain degree and the fair treatment of all bids received. Moreover, if money is put upfront, it is desirable that losing bids forfeits no money.

Franklin and Reiter [58] list in their paper some attacks sealed-bid auctions are subject to, classified into three classes. We have the attacks that deal with timing. We can exemplify them as the leakage when an internal collaborator opens the bids before the closure time and sends the information of the minimum value for a last-minute winning bid. Or an insider that manipulates the clock so that the auction closes prematurely or after the expected time, giving unfair advantage to some colluder. Another class of attacks concerns the integrity of the bid set, where the diversion of bids to other auctions closing earlier can give an unfair advantage so that the amounts are revealed and the subsequent auction can be won by a minimal price. Or the bid set can be manipulated to invalidate the wining bid so that the colluder can win. And the third class concerns the attacks after the winner is declared. In this scenario an insider can award the auction item to a person other than the winner. Or an insider collects payment for the losing bids, or even the winning bid defaults on the payment of the won bid and the auction must be re-run.

The protocol proposed by Franklin and Reiter tries to address the above listed attacks by introducing Byzantine fault tolerance, novel cryptographic techniques and novel security components such as electronic money. We say that a peer is correct if it follows the protocols, while if it is faulty nothing can be said about it. The properties claimed by them for their sealed-bid auction protocol are divided into two classes. The first class, called validity, contains the following claims:

1. The bidding phase will eventually close, but only after a correct auction server decides it should be closed.
2. There is at most one winning bid in each auction, which is proclaimed by applying the deterministic and publicly known rule to the correct bid received before the end of the bidding phase.
3. The action house collects payment equal to the amount of the winning bid from the winning bidder.
4. Correct losing bids should forfeit no money.
5. Only the winning bidder can collect the won item.

The second class of properties claimed by the authors regards secrecy and is shown below:

1. The identity of a correct bidder and the amount he bids for an item are not revealed to any party before the end of the bidding phase.

The authors also clearly state that the protocol is not resistant to price fixing by bidders, where they collude off-protocol to cap the winning bid. They do not guarantee that a bid will be included in the set of bids, since they have no control over interception or delay for message at the network layer.

Central to our verification efforts are some of the concepts over which the sealed-bid auction protocol was designed. As most of its claims concern validity properties, our verification will not be able to analyse some security properties we intended when choosing the protocol. Another important remark regarding the protocol design is the lack of guarantees to the bidders that their bids will be counted and that their electronic cash will not be misused.

### 5.1.2 Multicast Implementation Assumptions

The Franklin-Reiter protocol makes use of a series of different group multicast primitives. They were one of the motivations for our multicast events theory specification proposed on Chapter 3. Their definition of the different multicast primitives is slightly different from those standard ones we built the message casting framework upon. We will briefly cover them here.

The definition adopted by Franklin and Reiter [58] for unreliable multicast states that if a peer $S$ is correct then all members of the multicast group $G$ will receive the same sequence of unreliable multicasts from $S$. However, nothing can be said regarding a faulty $S$. Their reliable multicast definition provides an extension of unreliable multicast that delivers the same sequence of reliable multicasts to all members of $G$ regardless of the state of initiator $S$. However, reliable multicasts from different initiators can be received in different orders at each member of $G$. Their atomic multicast definition extends their reliable multicast one and strengthens it by ensuring that all correct members of $G$ receive exactly the same sequence of atomic multicasts regardless of the state of the sender. The guarantees provided to members and non-members of $G$ while in the initiator position for a multicast are the same. The only difference is practical, since failure detection in the case of atomic multicast cannot be achieved.

### 5.1.3 Electronic Money Abstraction

One of the novelties of the Franklin-Reiter protocol is the capability of verifying that the bids do not default prior to issuing the token for the winning bidder. To enable such capability, the protocol must make use of some form of digital cash scheme.

Digital cash schemes are sets of cryptographic protocols that enables a customer to withdraw money from a bank in digital form. The customer then uses it to purchase something, which enables the vendor to deposit the money in his account [42]. The objectives behind these digital cash schemes are to ensure that the identity of the customer is protected, providing a degree of anonymity. They must also enable parties only to accept valid digital money and that the money cannot be forged or reused.

Digital cash schemes can be divided into those that operate on-line, meaning they involve the active presence of the Bank to check the validity of the coins, and those said to be off-line, where the Bank only participates in the issuing and collecting phases [43]. To further exemplify this classification, in an on-line scheme the vendor can query the Bank for the validity of a coin regarding the property of it not being spent twice. Off-line schemes normally come with the costumer's identity embedded so that any double spending can be detected by the Bank when the second copy of the coin is paid in.

Franklin and Reiter use offline cash schemes. In their paper [58] they create an abstraction for digital cash that is claimed to be workable with most off-line schemes. They simply describe a digital coin as being a triple $(v_\$, \{|v_\$|\}_{Kr_{Bank}}, w_\$)$. In the triple, $v_\$$ is the coin's face description, $\{|v_\$|\}_{Kr_{Bank}}$ is the signature of the Bank that gives validity to this face description and $w_\$$ is some auxiliary information that must be present with the coin when it is used to a specific purchase.

The face description $v_\$$ will typically include the monetary value for the coin, as well as the identity of the customer protected by an anonymity function, as described above. The auxiliary information $w_\$$ is unique for any spending and will bring together with freshness some type of hint that enables the Bank or the vendor to reveal the identity of the double spender.

The Franklin-Reiter protocol requires the digital cash scheme to provide a function to deterministically determine locally the validity of $v_\$$ and $w_\$$. For our specification efforts, as we will show below, the coins will be treated only as nonces, and their validity will be deterministically accepted to follow protocol execution. We will not address the presence of the identity revealing bits on $w_\$$, and $v_\$$. We assume coin components to be unique and non guessable, thus helping on the determinism of finding the winner.

### 5.1.4 Verifiable Signature Sharing

Verifiable signature sharing [57], or simply $V\Sigma S$, is a primitive that enables the holder of a digitally signed message to share the signature among a group of peers so that they can reconstruct the signature later, similarly to verifiable secret sharing primitives. At the end of the sharing phase, the members can verify that they possess a valid share and that the signature can be reconstructed even if the original signer or some trustees are faulty. Faulty trustees gain no information regarding the original signature.

Although the Franklin and Reiter paper stresses the importance of the $V\Sigma S$ to their specification, in terms of our abstraction, $V\Sigma S$ will be considered as a specific case of verifiable secret sharing. Their concerns regarding the usage of verifiable secret sharing in opposition to $V\Sigma S$ are vague except for the technicalities presented. We nevertheless recognise the existence and importance of such primitive to deal with signed documents that need to be valid only in the future and the control of this release should be subject to an agreement. Good examples of such are a will, or a "springing power of attorney".

The distinction between $V\Sigma S$ and verifiable secret sharing is mostly technical, and their properties are very similar. In its ability of checking integrity of a secret that was shared without the necessity of reconstruction, the $V\Sigma S$ extends a verifiable secret sharing scheme by broadcasting the result of its own computation.

### 5.1.5 Protocol Description

The protocol proposed by Franklin and Reiter is constructed using $n$ auction servers, of which $t$ are assumed to operate faithfully. The parameter $t$ is the threshold for the fault tolerance mechanism. A maximum of $t$ servers can be compromised and the service can still run securely. In its conception the protocol is claimed to be Byzantine-failure secure.

A bidder submits a bid with the amount he wants to pay for the item by sharing a digital coin $(v_\$, \{|v_\$|\}_{Kr_{Bank}}, w_\$)$ with this value among all servers hosting the auction. To avoid the auction servers cheating, the coin values are split in different ways. The values of $v_\$$ and $w_\$$ are split using a standard secret sharing mechanism using our fault tolerance value of $t$ as threshold. The signature of the face value for the coin $\{|v_\$|\}_{Kr_{Bank}}$ is shared using a $V\Sigma S$ also using $t$ as threshold.

Once the bidding phase finishes, the servers in agreement will reconstruct the values for $v_\$$ and $w_\$$ for all bids cast during the bidding phase and will independently determine the winner. For the winning bid, the auction servers will perform a $V\Sigma S$ verification to see if the bid is valid and the money can be collected by reconstructing $\{|v_\$|\}_{Kr_{Bank}}$. After this verification, the auction servers can award a token to the winning bidder to collect the item.

The original paper proposing the protocol [57] is difficult to parse and understand. Also, there are a lot of implicit calculations that are to be assumed by the specification.

This is clearly a violation of the protocol design principles we discussed on Chapter 2, which introduces some vulnerabilities. Nevertheless, we summarise the description of the Franklin-Reiter sealed-bid auction protocol in Figure 5.1.

$$\begin{array}{llcll}
1. & B & \overset{A}{\rightsquigarrow} & SG & : & \lambda X.\{|aid, \{|S(B, v\$, w\$)_X, aid|\}_{Kr_X}, \\
 & & & & & V\Sigma S\_pub(\{|v\$|\}_{Kr_{Bnk}}), \\
 & & & & & \{|V\Sigma S\_priv(\{|v\$|\}_{Kr_{Bnk}})_X|\}_{Kr_X}|\} \\
2. & S_i & \overset{A}{\rightsquigarrow} & SG & : & aid, close \\
3. & S_i & \overset{U}{\rightsquigarrow} & SG & : & \lambda XY.\{|aid, S(Y, v\$, w\$)_X|\} \\
4. & S_i & \overset{R}{\rightsquigarrow} & SG & : & \lambda X.aid, V\Sigma S\_stat(\{|v\$|\}_{Kr_{Bnk}})_X \\
5. & S_i & \rightarrow & B & : & aid, B, \{|aid, B|\}_{Kr_{S_i}}
\end{array}$$

Figure 5.1: Franklin-Reiter Sealed-Bid Auction Protocol

The protocol execution starts with the bid casting phase by the issuing of message one by the bidder. The bidder $B$ in possession of a digital coin $(v_\$, \{|v_\$|\}_{Kr_{Bank}}, w_\$)$ will issue an atomic multicast to the multicast group $SG$ comprised of all the auction servers participating of the auction. This multicast message starts with the auction identification token *aid*. This will be followed by the $n, t$-sharing of the concatenation of his identity, the face value of the coin, and the freshness value for the coin. Each share is encrypted to the public key of each corresponding server in the multicast group $SG$. This share will be followed by the public $V\Sigma S$ of the Bank's signature to the coin's face $V\Sigma S\_pub(\{|v\$|\}_{Kr_{Bnk}})$, and the private $V\Sigma S$ $n, t$-shared to all members of $SG$. The $V\Sigma S$-private shares will be encrypted with the public key of each corresponding server in the multicast group $SG$.

After the bid casting phase finishes, each auction server $S_i$ in the multicast group $SG$ will multicast the second message to the group. This message simply states the auction identification and the closing statement. After each auction server $S_i$ has received at least $t$ atomic multicasts stating the bidding phase is closed, no more bids are accepted. The inclusion of this message in the protocol is controversial for having no security, but the authors argue that due the implementation characteristics of atomic multicast, communication is authenticated within the multicast group.

After the closure of the bidding phase we start the bid opening phase. Each auction server $S_i$ will multicast to the multicast group $SG$ the auction identification *aid* and his shares $S(Y, v\$, w\$)_X$ composed of the concatenation of the bidder's identity, the face value of the coin and the freshness function of the coin. After the reception of $t$ multicasts, a server can locally reconstruct each bid $Y, v\$, w\$$ and deterministically compute the winner.

With the winner locally determined, each auction server $S_i$ will reliably multicast to the multicast group $SG$ message four, which is composed of *aid* and the result of the $V\Sigma S$ verification of his share for the winning bid for the bank's signature to the coin's face value $v\$$. After the reception of $t$ multicast messages an auction server can locally decide whether the winning bid is valid.

The winner declaration phase consists of each auction server $S_i$ issuing a unicast message to the winning bidder $B$. The message is composed of *aid*, the bidder's identity and the signature of the concatenation of these values by the auction server $S_i$. A bidder can collect the won item if he possesses $t$ tokens signed by different auction servers.

Although the protocol takes care of checking the bank's signature in the coin, coin reconstruction and deposit is out of scope for the protocol. By using an off-line digital cash scheme, the protocol provides a degree of anonymity against the detection of the spending by the Bank. The authors propose an anonymity scheme to protect the bidder's anonymity against the auction house.

### 5.1.6 Enabling Anonymity

One interesting optional feature added to the Franklin-Reiter sealed-bid auction protocol is that of accepting anonymous bids. As it already uses electronic money to provide anonymity for the spending against the Bank which issued the coin, a desirable extension would be to provide anonymity in the bidding against the auction house.

To prevent the release of the bidder's identity to the auction house we simply substitute the identity of the bidder $B$ by the hash of a large random number $H(r)$ generated and known only to the bidder. This pseudonym will only impact in the modification of messages one and five of the protocol as shown in Figure 5.2. To claim the item won by the winner declaration message five, the bidder must show that he posseses the random number $r$ that will produce the pseudonym $H(r)$.

$$\boxed{\begin{array}{llcllc}
1. & B & \overset{A}{\rightsquigarrow} & SG & : & \lambda X.\{|aid, \{|S(H(r), v\$, w\$)_X, aid|\}_{Kr_X}, \\
 & & & & & V\Sigma S\_pub(\{|v\$|\}_{Kr_{Bnk}}), \\
 & & & & & \{|V\Sigma S\_priv(\{|v\$|\}_{Kr_{Bnk}})_X|\}_{Kr_X}|\} \\
2. & S_i & \overset{A}{\rightsquigarrow} & SG & : & aid, close \\
3. & S_i & \overset{U}{\rightsquigarrow} & SG & : & \lambda XY.\{|aid, Y, v\$, w\$)_X|\} \\
4. & S_i & \overset{R}{\rightsquigarrow} & SG & : & \lambda X.\{|aid, V\Sigma S\_stat(\{|v\$|\}_{Kr_{Bnk}})_X|\}_{Kr_X}|\} \\
5. & S_i & \overset{B}{\rightsquigarrow} & ALL & : & aid, H(r), \{|aid, H(r)|\}_{Kr_{S_i}}
\end{array}}$$

Figure 5.2: Franklin-Reiter Sealed-Bid Auction Protocol with Anonymity

Message one in the Franklin-Reiter protocol with anonymity starts with the bidder $B$ possessing a digital coin $(v_{\$}, \{|v_{\$}|\}_{Kr_{Bank}}, w_{\$})$ and being able to generate a long random number $r$ to be hashed and used as his pseudonym. He will then issue an atomic multicast to the multicast group $SG$. His multicast message starts with *aid* followed by the $n, t$-sharing of the concatenation of his pseudonym $H(r)$, the face value of the coin, and the freshness value with a reconstruction trigger $t$, encrypting each of the shares to the public key corresponding to each server in the auction. This is followed by the public $V\Sigma S$ of the bank's signature to the coin's face $V\Sigma S\_pub(\{|v\$|\}_{Kr_{Bnk}})$, and the private $V\Sigma S$ $n, t$-shared to all members of $SG$ and encrypted as the same form of the other values.

Messages two, three and four remain the same. Message five will be converted into a Broadcast from each of the auction servers $S_i$. The message is composed by *aid*, the bidders pseudonym $H(r)$ and the signature of the concatenation of these by the auction server $S_i$. A bidder can collect the item if he possesses different signed tokens from more than $t$ auction servers and if he can show he knows $r$ so that the winner's pseudonym is $H(r)$.

While this measure seems to ensure that the auction server cannot identify the bidder, some other steps are needed to guarantee that. Special attention should be drawn to the digital cash scheme being used. Each coin embeds a function of the identity of the costumer that can be obtained in the case of any double spending. The main precaution $B$ can take to avoid being identified is to never reuse any coin. This scheme provides only a weak form of anonymity since any collusion between the auction house and the Bank can reveal the identity of the bidder in most off-line digital cash schemes.

Although the idea of verifying anonymity properties is interesting, we decided not to pursue it. The security goals claimed for the protocol are in general weak and do not require the full capabilities of the method. It was also not clear at the time of the development that anonymity could be stated in a meaningful way.

### 5.1.7 Known Issues

The protocol proposed by Franklin and Reiter puts little effort into providing guarantees to all parties involved in the protocol execution. For the bidder, it provides no confirmation that the bid was included in the auction or that the bid lost the auction. The author's justification is the use of atomic multicast. There is no explanation why the losing bids are not notified, which makes the protocols unbalanced in terms of what is available to each participating peer.

The authors considered a threshold signature scheme in which the cooperation of $t$ servers would be required to create the token representing the winner declaration, but they dropped this implementation on the basis of complexity. This would have definitely created a stronger version than a series of unicast. Their justification for opting out for this scheme was based on the complexity of the operations with the available primitives to enable such joint signatures. This would make the implementation of the system much more computationally intensive, but at the same time would have given a full Byzantine-failure safety to the protocol.

Another important issue that can be raised regarding the protocol, even before any formal analysis of it, is the timeliness of the coin reconstruction. Problems regarding the coin validity can arise between the time it was verified during the protocol execution and the coin is paid to the vendor's account, for example if the coin is used in another auction. The authors note that the bidder can be identified for double spending, but this does not avoid the fact that no money can be yielded to pay for the delivered item.

## 5.2 Modelling the Franklin-Reiter Auction Protocol

After reviewing the protocol threat model and requirements, we start with the verification process by first specifying the protocols using the inductive method and the tool Isabelle/HOL.

### 5.2.1 Auxiliary Definitions

To be able to specify the Franklin-Reiter protocol we need some auxiliary functions. The first set of auxiliary functions we had to specify were those that made the specification of some cryptographic operations simpler. As shown before in Chapter 2 the inductive method uses *Crypt* to handle symmetric and asymmetric cryptographic operations. It also has the definition for *Hash*. The Franklin-Reiter protocol makes use of digital signatures, we decided to specify some syntactic translations for it. We borrowed this from Bella [20] as shown on Definition 46.

**Definition 46.** *Definition of translations for signatures*

**constdefs**
```
sign ::  "[key, msg] => msg"
"sign K X == {| X, Crypt K (Hash X) |}"
signOnly ::  "[key, msg]=>msg"
"signOnly K X == Crypt K (Hash X)"
signCert ::  "[key, msg]=>msg"
"signCert K X == {|X, Crypt K X |}"
```

Another important auxiliary definition concerns the creation of bid sets for all the parallel auctions. When specifying the Frankiln-Reiter protocol, we noticed that the shared token $S(B, v\$, w\$)$ did not contain any information related to the auction taking place. This makes the identification of bid replays very difficult. We defined the function *bids* which takes as arguments *aid* and a trace of events *evs* and returns a set of triples containing the bids as shown in Definition 47.

**Definition 47.** *Definition of function bids*

**constdefs**
```
bids ::  "[nat,event list] ==> (agent*nat*nat) set"

"bids aid evs == { (B, v, w).  ∃ multicast_group t Bank.
     Multicast B multicast_group (λC. {|Nonce aid,
          Crypt (pubK C)({|Nonce(share(nat t,multicast_group,C)
                    {|Agent B, Nonce v, Nonce w|}), Nonce aid|}),
          Nonce ( pub_share (nat t,multicast_group,C)
                    (signOnly (priSK Bank) (Nonce v))),
          Crypt (pubK C)(Nonce(priv_share(nat t,multicast_group,C)
                    (signOnly(priSK Bank)(Nonce v))))|})∈ set evs}"
```

The specification of the function *bids* simply yields the set of all bids for an auction by extracting the required information from the syntax of message one. It states that all triples of bidder identity, coin face value and coin freshness values are included in the set for that auction identification for all multicast groups, thresholds and Banks present in the syntax of message one.

The third and final auxiliary definition is called *keyfree*. During the verification of secrecy for the Franklin-Reiter protocol, we often were faced with sub goals with the form $X \in$ *analz* $(G \cup H)$. These goals are notoriously complex to address since they trigger rule *Decrypt* from the *analz* function's definition (Definition 17). The inductive function *keyfree* is defined by the set of all messages that contains no keys. That is, all atomic items belong to it and we can combine key-free messages to create bigger ones as shown in Definition 48.

**Definition 48.** *Definition of keyfree*

**inductive_set**
```
keyfree ::  "msg set"
   where
   Agent:  "Agent A ∈ keyfree"
   | Number:  "Number N ∈ keyfree"
   | Nonce:  "Nonce N ∈ keyfree"
   | Hash:  "Hash X ∈ keyfree"
   | MPair:  "[|X ∈ keyfree; Y ∈ keyfree|] ==> {|X,Y|} ∈ keyfree"
   | Crypt:  "[|X ∈ keyfree|] ==> Crypt K X ∈ keyfree"
```

The definition of *keyfree* is important because we can divide the set from function *analz* into those having the constructor *Key* and those not having it. This enables us to prove [|*X* ∈ *analz (G* ∪ *H); G* ⊂ *keyfree*|] ==>*(X* ∈ *parts G* ∪ *analz H*). This makes contradictions easier since we can find them now over *parts* instead of *analz*.

### 5.2.2 Protocol Specification

The specification of the Franklin-Reiter Sealed-bid auction protocol starts, as usual, with the definition of a constant naming the specification called *fr*. It is an inductive set of lists of events which will represent the formal protocol model. We define the empty trace by the rule *Nil* which sets the base of the induction. The Isabelle/HOL specification is shown on Definition 49.

This protocol implements communication using multicast and unicast. For our purpose, we opted for using separate primitives instead of using the multicast primitive in unicast mode of operation as shown in Chapter 3. For that we had to formalise the Spy's illegal activity twice: one when he is acting under the unicast framework and another when he is acting under the multicast framework. These rules enable the *Spy* to perform his interferences in the protocol using both communication methods.

Rule *Fake* is the classical rule for Spy's interference in the protocol which states that the *Spy* is able to *Say* to any peer *B* the synthesis of all he can analyse from the trace of events. Rule *FakeMC* is a multicast variant of the rule *Fake* which gives the *Spy* the power to multicast any message with components he learnt from traffic analysis to any multicast group of the protocol.

**Definition 49.** *Inductive definition of Franklin-Reiter Protocol - Basic Steps*

**inductive_set** `fr ::  "event list set"`
```
where
  Nil:  "[] ∈ fr"

| Fake:  "[| evsf ∈ fr;
           X ∈ synth (analz (knows Spy evsf)) |]
           ==> Says Spy B X # evsf ∈ fr"

| Fakemc:  "[| evsfmc ∈ fr;
             XF ∈ synth (analz (knows Spy evsfmc)) |]
     ==> Multicast Spy multicast_group (λC. XF) # evsfmc ∈ fr"

| Reception:  "[| evsr ∈ fr; Says A B X ∈ set evsr |]
```

```
                 ==> Gets B X # evsr ∈ fr"

| ReceptionMC: "[| evsrmc ∈ fr;
            Multicast A multicast_group (λC. XF ) ∈ set evsrmc;
            B ∈ set multicast_group |]
            ==> Gets B XF # evsrmc ∈ fr"
```

Although our initial specification took into account the reception of messages, due to the characteristics of the protocol and the lack of proper information distribution to all peers, its usage was hindered. The protocol provides very little information to the bidder and due to the intrinsic characteristics of the communication and goals we were verifying, we did not use the message reception framework in any lemma. Nevertheless we implemented them here for the sake of completion.

Rule *Reception* is the classical inductive method's reception rule which states that a *Says* event was casted with $B$ as its receiver, the event *Gets* to $B$ for that message is called. Analogously, rule *ReceptionMC* states that if $B$ belong to a multicast group that is the destination of a *Multicast* event from $A$, the event *Gets* to $B$ is called as a function of $XF$ to $B$. This will confirm the reception of the multicast by agent $B$. Some tests we did with message reception under multicast show the potential of the inductive method in managing big loads of information. The specification of *ReceptionMC* clearly shows the exponential explosion in terms of knowledge distribution when multicasting. To each *Multicast* event, we have potentially infinite *Gets* events.

Still regarding to *ReceptionMC* event, one of our options in implementing the Franklin-Reiter protocol was to assume all multicasts were atomic. By assuming that, no message reception need to be performed because we assume all peers receive the message totally when it was sent. This choice was based on simplifying the model.

#### 5.2.2.1 Bid Casting

With the basic inductive method's specification in place, we start now describing the protocol messages by first capturing the bid casting phase. The specification of the first message of the bid casting phase is shown below on Definition 50. Some of its preconditions are merely technical.

**Definition 50.** *Inductive definition of Franklin-Reiter Protocol : Bid Casting*

```
| FR1:  "[| evs1 ∈ fr; Nonce w ∉ used evs1; w ≠ close;
            w ∉ sessionIDs; w ∉ shares;
            aid ∈ sessionIDs; v ≠ close; v ∉ sessionIDs;
            v ∉ shares; Nonce v ∉ used evs1;
            Multicast S multicast_group (λC. {|
                       Nonce aid, Number close|}) ∉ set evs1 |]
==> Multicast B multicast_group (λC. {|Nonce aid,
      Crypt (pubK C) (
           {|Nonce (share (nat t, multicast_group, C) {|
            Agent B, Nonce v, Nonce w|}), Nonce aid|}),
      Nonce ( pub_share (nat t, multicast_group, C)
            (signOnly (priSK Bank) (Nonce v))),
      Crypt (pubK C )(
           Nonce ( priv_share (nat t, multicast_group, C)
           (signOnly (priSK Bank) (Nonce v))) )|})
 # Notes B {|Nonce aid, Nonce w, Nonce v|} # evs1 ∈ fr"
```

Definition 50 starts with inductive rule *FR1*, stating that the trace *evs1* belongs to the inductive set *fr*, the nonce *w* and the nonce *v* were not used in this trace before, are not equal to the constant *close*, does not belong to *sessionIDs* and does not belong to *shares*. We also require that the auction identification *aid* belongs to *sessionIDs* and that message two closing the auction has not appeared in the trace. If these preconditions are met, we extend the trace of events *evs1* belonging to *fr* with message one.

Message one is a *Multicast* event from the bidder *B* to the multicast group of auction servers. Its payload has the nonce *aid* as the session identification, followed by the bidder *B*'s identity concatenated with the digital coin *v*, and nonce *w*. The bidder's identity, coin description and coin function are shared to the multicast's destinations with a threshold *t*. Each share is encrypted with the intended destination's key creating the sharing token *share (nat t, multicast_group, C)* {| *Agent B, Nonce v, Nonce w*|} which is shielded by a nonce type before being encrypted. Then the coin's digital signature's public share is included using a verifiable secret sharing scheme to the same group and threshold parameters as the initial sharing scheme. The public part *pub_share (nat t, multicast_group, C)(signOnly (priSK Bank)(Nonce v))* is shielded by a nonce type and added to the message straight away. The private shares are encrypted with the intended destination's key creating the sharing token *priv_share (nat t, multicast_group, C) (signOnly (priSK Bank) (Nonce v))* this is shielded by a nonce type before being encrypted.

To finish the *FR1* specification, we give the bidder *B* the knowledge of what he included in the shares trough an event *Notes*. This is needed because *parts* and *used* do not know about *share*. For this reason we need to explicitly give to *B* this knowledge following our secret sharing specification shown on Chapter 4.

We must call to attention two important details of our specification. The first is the fact that we don't check freshness of *aid*. This happens because we want message FR1 to repeat for later bids. The second specification detail is that we only allow *FR1* to appear if no *FR2* was ever casted. We try to preserve the consistency of the bid set after auction closure. This does not act with fidelity with the protocol operation, since this should only be happening after the threshold of messages was received. Again we opted for the simplest specification, taking into account properties of atomic multicast and the assumption that all servers following the protocol would close the auction at exactly the same time. So the impact of these in this initial verification we conducted is marginal.

#### 5.2.2.2 Bid Closure

After the bid casting phase, we have an event that indicates the closure of that phase. Message two is specified as rule *FR2* in our specification of the Franklin-Reiter protocol.

The preconditions of rule *FR2* are that the trace of events *evs2* extends the inductive set *fr* for the protocol specification, that *aid* belongs to the set of *sessionIDs*, the auction server *S* belongs to the multicast group and that we have at least one casted bid. This is represented by the existence of message one in the trace of events. If these preconditions are met we extend the trace of events *evs2* with a multicast from the *S* to the multicast group containing *aid* and the command *close* as shown in Definition 51.

**Definition 51.** *Inductive definition of Franklin-Reiter Protocol : Bid Closure*

```
| FR2:  "[|evs2 ∈ fr; S ∈ set multicast_group; aid ∈ sessionIDs;
          Multicast B multicast_group (λC. {|Nonce aid,
          Crypt (pubK C) (
               {|Nonce (share (nat t, multicast_group, C) {|
               Agent B, Nonce v, Nonce w|}), Nonce aid|}),
          Nonce (pub_share (nat t, multicast_group, C)
               (signOnly (priSK Bank) (Nonce v))),
          Crypt (pubK C )(
               Nonce ( priv_share (nat t, multicast_group, C)
               (signOnly (priSK Bank) (Nonce v))))|})
               ∈ set evs2 |]
==> Multicast S multicast_group (λC.{|Nonce aid, Number close|})
    # evs2 ∈ fr"
```

Again in message two we had to take some decisions regarding our specification. Similarly to message one, we did not implement any trigger and assumed the multicast as being atomic. But here it is evident by a very strong assumption from the authors regarding their implementation of atomic multicast. As we explained above (§5.1.2) one of their assumptions was that multicasts from within the group are authenticated. But as our specification follows a broader definition of atomic multicast this was not available in the specification we proposed. We took this specification decision bearing in mind that this could clearly reveal some attacks the authors would refute on the basis of their very strong assumption.

#### 5.2.2.3 Bid Opening

Once no more bids can be cast in the auction and the auction servers agree that the auction is closed, we should proceed to a phase where we open the bids to verify which bidder cast the best bid. In the Franklin-Reiter protocol this is done by message three, which is specified by our inductive rule *FR3* as shown in Definition 52.

**Definition 52.** *Inductive definition of Franklin-Reiter Protocol : Bid Opening*

```
| FR3:  "[| evs3 ∈ fr ; S ∈ set multicast_group ; S ∉ bad;
          aid ∈ sessionIDs; (B, v, w): bids aid evs3;
          Multicast S multicast_group (λC.
               {| Nonce aid, Number close|}) ∈ set evs3 |]
==> Notes S {|Nonce aid, Nonce w, Nonce v|} #
    Multicast S multicast_group (λC. {|
     Nonce aid,
     Nonce (share (nat t, multicast_group, S)
          {|Agent B, Nonce v, Nonce w|})|}) # evs3 ∈ fr"
```

The preconditions for rule *FR3* are that the trace of events *evs3* is part of our inductive specification for the protocol, that the auction server $S$ is not compromised, that the auction identifier *aid* is a session identification, that the triple *(B, v, w)* represents a valid bid for the auction *aid* in the events trace *evs3* and that the auction was closed by the existence of message two in the trace.

If these preconditions are met we extend the trace *evs3* with two events. A first event *Notes* to the auction server *S* the values for *aid*, *v* and *w* followed by a *Multicast* event from the auction server *S* to the multicast group running the auction service. The payload of this message is composed of *aid*, plus the share the server holds for the bid being broadcasted.

In the specification of rule *FR3* we have also to stress some specification decisions. A clear one is again the presence of our knowledge distribution scheme for the information that was shared. In rule *FR3* we extend the trace by a *Multicast* event and then we extend it again with a *Notes* events. This is done for the sake of giving the servers following the protocol the knowledge regarding the values of the bids they opened. Again we oversimplify our specification by not implementing any triggered reconstruction and assume all multicasts to be atomic. We also do not provide any guarantee that all bids will be opened, which would be difficult to do with the inductive method because we cannot force events to happen.

#### 5.2.2.4 Bid Validation

Once the winning bid is found by all auction servers they will start the bid validation phase. We will check the validity of the digital coin submitted with the bid. Message four is represented in our specification by rule *FR4* as shown on Definition 53.

The preconditions for firing rule *FR4* are that the trace of events *evs4* is part of the inductive set describing the protocol specification, that *aid* is in the set of *sessionIDs*, that the triple *(B, v, w)* is a valid bid within the auction *aid* in the trace *evs4* and that the nonces *w* and *v* are in the knowledge of the auction server *S*. If these preconditions are met we extend the trace of events *evs4* by adding an event *Notes* to the auction server *S* for the bank's digital signature for the coin's face value. Then we add a *Multicast* event from server *S* to the multicast group of servers running the auction containing *aid* and his share of the digital signature for the coin of the winning bid.

**Definition 53.** *Inductive definition of Franklin-Reiter Protocol : Bid Collection*

```
| FR4:  "[| evs4 ∈ fr; S ∈ set multicast_group ;
           aid ∈ sessionIDs; (B, v, w):  bids aid evs4; S ∉ bad;
           Nonce w ∈ knows S evs4; Nonce v ∈ knows S evs4; |]
==> Notes S (signOnly (priSK Bank) (Nonce v)) #
    Multicast S multicast_group (λC. {|
     Nonce aid,
     Nonce ( priv_share (nat t, multicast_group, S)
            (signOnly (priSK Bank) (Nonce v)))|}) # evs4 ∈ fr"
```

On rule *FR4*'s specification we have the same scenario as with *FR3* where we have a double extension of the trace representing that all servers multicast their shares and that they were able to reconstruct the signature of the coin's face value. Here we took another simplification step within the original protocol description. Instead of implementing the verifiable signature sharing proposed by the authors we decided to implement a reconstruction using the casting of the private shares to the multicast group members.

This specification choice clearly weakened the protocol, since at this point one server can collude with the attacker and deposit the coin for himself. But our choice is justified beyond the plain simplification of the verification process, since the guarantees the original protocol yields are just that the coin may be reconstructable at a later time, and not that it is indeed reconstructable, since more than $n - t$ server can be corrupted after the protocol run.

Finally the justification for the changes in the rule *FR4* is based on the fact that we shifted our focus to the verification of the suitability of the multicast theory instead of the full verification of the Franklin-Reiter protocol. We focused only in achieving the verification of the secrecy property for the bids before closure time. This was enough to validate the multicast events theory.

#### 5.2.2.5 Winner Declaration

With the winning bid known and with its digital coin payable, we can now deliver to the winner the tokens he needs to collect the item. Message five is a unicast from each one of the servers that concluded the protocol execution to the winning bidder with the winner declaration token. We specify message five from the Franklin-Reiter protocol as our inductive rule *FR5* .

The specification of rule *FR5* starts with the precondition that the trace of events *evs5* is part of the inductive set describing the protocol specification, that *aid* is in the set of *sessionIDs*, that the triple *(B, v, w)* is a valid bid within the auction *aid* in the trace *evs5* and that the nonces *w*, *v* and the digital signature for the coin's face value *(signOnly (priSK Bank) (Nonce v))* are in the knowledge of the auction server *S*. If these preconditions are met, the auction server *S* issues a unicast with the event *Says* to the winning bidder *B*. The payload of this message is *aid*, the winners identification *B* and the signature using the server's private signature key to the previous content.

**Definition 54.** *Inductive definition of Franklin-Reiter Protocol : Winner Declaration*

```
| FR5:  "[|evs5 ∈ fr; S ∈ set multicast_group; aid ∈ sessionIDs;
          S ∉ bad; w ∉ sessionIDs;
          w ∉ shares; (B, v, w):  bids aid evs5;
          Nonce w ∈ knows S evs5; Nonce v ∈ knows S evs5;
          (signOnly (priSK Bank) (Nonce v)) ∈ knows S evs5|]
==> Says S B {| Nonce aid, Agent B,
          sign (priSK S) {| Nonce aid, Agent B |}|} # evs5 ∈ fr"
```

With the specification of message five we complete the protocol description. Note that the specification of rule *FR5* deliberately takes some steps to test our multicast specification and the distribution of knowledge within the protocol. First, the pre-conditions to the firing of *FR5* are based on the knowledge of the auction server *S* acquired during the previous phases and the contents of the bid set. Second, we deliberately did not represent the unicast method using the *Multicast* event to be able to test the integration of our specification with the original one in the inductive method.

Here we took similar specification decisions as we did in the previous messages. But note that the trigger required for collecting the item in this message is left off-protocol by the authors, which makes it difficult to verify the ability of collecting the item by the winner.

With the basic specification for the Franklin-Reiter sealed-bid auction protocol in place we will follow on doing the verification of the secrecy property for the bid casting phase. But before doing that we will create some support lemmas for the auxiliary definitions.

## 5.3 Verifying Frank-Reiter Auction Protocol

We start first with the proofs regarding the auxiliary functions we defined to use with the protocol specification. We proved some lemmas regarding function *bids* and *keyfree* to verify their correct specification and to help us with the general validity proof regarding the protocol specification.

### 5.3.1 Proofs regarding the set of bids

The lemma regarding the *bids* set establishes that to extend the set of bids we need the existence of messages with syntax of the rule FR1 in the trace of events. In a more pragmatic reading, it states that for all *B*, *multicast_group*, *t*, *Bank*, *v* and *w*, the head of the list of events different of message one implies that the set of bids of the trace of events is equal to the set of bids of the tail of the trace of events. Elements different from the *Multicast* on message one does not contribute to the list of bids. Its definition is shown in Lemma 46 (bids_not_insert).

**Lemma 46.** *bids_not_insert*

```
∀ B multicast_group t Bank v w.
    ev ≠ Multicast B multicast_group (λC. {|
       Nonce aid,
       Crypt (pubK C) ({|Nonce (share (nat t, multicast_group, C)
                        {|Agent B, Nonce v, Nonce w|}),
                        Nonce aid|}),
       Nonce ( pub_share (nat t, multicast_group, C)
                        (signOnly (priSK Bank) (Nonce v))),
       Crypt (pubK C)(Nonce(priv_share (nat t, multicast_group, C)
                        (signOnly (priSK Bank) (Nonce v))))|}))
==> bids aid (ev # evs) = bids aid evs
```

The proof for Lemma 46 is straightforward and only requires unfolding definition for bids and the call to the simplifier and classical reasoner. This fact is important to eliminate messages that do not yield bids.

The next lemma regards the insertion of the triples *(B, v, w)* in the set of bids for a given *aid*. If the set of bids for an auction *aid* in a trace with the head being the event yielded by rule *FR1*, this is equal to inserting the triple *(B, v, w)* to the set of bids for that trace. Its definition is shown in Lemma 47 (bids_insert).

**Lemma 47.** *bids_insert*

```
bids aid
(Multicast B multicast_group (λC. {|
      Nonce aid,
      Crypt (pubK C)({|Nonce (share (nat t, multicast_group, C)
                      {|Agent B, Nonce v, Nonce w|}),
                      Nonce aid|}),
      Nonce ( pub_share (nat t, multicast_group, C)
                      (signOnly (priSK Bank) (Nonce v))),
      Crypt (pubK C)(Nonce(priv_share (nat t, multicast_group, C)
                      (signOnly (priSK Bank) (Nonce v))))|}))#evs)
= insert (B, v, w) (bids aid evs)
```

Proving Lemma 47 (bids_insert) consists in calling the simplifier extended with the definition for *bids* followed by the classical reasoner augmented with the function congruence lemma to process the function in the multicast message. This fact is important to reason about the equality between the trace and the triples representing the bid for a given auction in a trace.

The next lemma regards the integrity of the *bids* set and the relation between the coin and the bidder, especially that the bidder can cast a single valid bid to the auction. Lemma 48 (bids_simp) states that a triple *(B, v, w)* belonging to the set of bids of an auction *aid* in a trace of events headed by the event yielded by rule *FR1* implies that the coin's face value and freshness information are equal to those in the triple, or the triple is in the tail of the trace.

**Lemma 48.** *bids_simp*

```
( B, v, w ) ∈ bids aid
(Multicast B multicast_group (λC. {|
     Nonce aid,
     Crypt (pubK C)({|Nonce (share (nat t, multicast_group, C)
                     {|Agent B, Nonce v', Nonce w'|}),
                     Nonce aid|}),
     Nonce ( pub_share (nat t, multicast_group, C)
                     (signOnly (priSK Bank) (Nonce v'))),
     Crypt (pubK C)(Nonce(priv_share (nat t, multicast_group, C)
                     (signOnly (priSK Bank) (Nonce v'))))|}))#evs)
==> (v = v' ∧ w = w')| ( B, v, w ) ∈ bids aid evs"
```

Proving Lemma 48 (bids_simp) is very similar to the proof of Lemma 47 (bids_insert), and consists in calling the simplifier extended with the definition for *bids* followed by the classical reasoner augmented with the function congruence lemma to process the function in the multicast message. This fact is important because it yields that a bidder can only cast a single valid bid and that the digital cash information are tied to the bidder in the *bids* set.

The next lemma lets us infer if there exists an event yielded by rule *FR1* in the trace of events, there will be a bid in the bid set for that message in the same trace. More pragmatically the existence of the event *Multicast* yielded by *FR1* implies that the triple *(B, v, w)* is in the set *bids* for the *aid* mentioned in the message for the given trace. This is shown in Lemma 49 (bidsI).

**Lemma 49.** *bidsI*

```
Multicast B multicast_group (λC. {|
      Nonce aid,
      Crypt (pubK C)({|Nonce (share (nat t, multicast_group, C)
                 {|Agent B, Nonce v, Nonce w|}),
                 Nonce aid|}),
      Nonce ( pub_share (nat t, multicast_group, C)
                 (signOnly (priSK Bank) (Nonce v))),
      Crypt (pubK C)(Nonce(priv_share (nat t, multicast_group, C)
                 (signOnly (priSK Bank)(Nonce v))))|})∈ set evs)
==> ( B, v, w ) ∈ bids aid evs"
```

Proving Lemma 49 (bidsI) consists of unfolding the *bids* definitions and using the tactic *auto*. The importance of this lemma regards the fact that it yields the direct relation between the presence of message one of the protocol to a bid being in the *bids* set for an auction *aid*. This fact is generally used when we want to reason about the *bids* set from content of the trace.

Lemma 50 (bidsM) is the converse of Lemma 49 (bidsI). It shows the other direction of the implication regarding the event yielded by *FR1* and the presence of a bid triple in the *bids* set. It states that if a triple *(B, v, w)* is in the *bids* set for an auction *aid*, this implies that there exists a *multicast_group*, a *t* and a *Bank* so that an event *Multicast* with the syntax of the one yielded by rule *FR1* is in the trace of events with the parameters of the given triple.

**Lemma 50.** *bidsM*

```
( B, v, w ) ∈ bids aid evs ==> ∃ multicast_group t Bank.
Multicast B multicast_group (λC. {|
      Nonce aid,
      Crypt (pubK C)({|Nonce (share (nat t, multicast_group, C)
                 {|Agent B, Nonce v, Nonce w|}),
                 Nonce aid|}),
      Nonce ( pub_share (nat t, multicast_group, C)
                 (signOnly (priSK Bank) (Nonce v))),
      Crypt (pubK C)(Nonce(priv_share (nat t, multicast_group, C)
                 (signOnly (priSK Bank)(Nonce v))))|})∈ set evs)
```

Proof for Lemma 50 (bidsM) requires the call for the classical reasoner augmented with the definition of *bids* and the function congruence fact for dealing with the $\lambda$ function in the *Multicast* event. The fact yielded by Lemma 50 (bidsM) is important we want to reintroduce the event that generated a bid triple in the *bids* set for a given auction.

### 5.3.2 Proofs regarding keyfree

As we explained above, the definition of function *keyfree* helps us to reason about messages that do not include key material and help us to simplify arguments to *analz*. To make it usable we had to prove a couple lemmas, in special those that helps us simplify the argument of *analz* into *parts* because there are no keys in $G$ to help decrypt messages in $H$.

We start with Lemma 51 which is proven by appealing to the inductive definition of *parts* and the tactic *auto*. We will use it later to prove Lemma 52

**Lemma 51.** *parts_keyfree*

```
parts (keyfree) ⊆ keyfree
```

An important lemma that helps us to transform complex form of *analz* into simpler instances of *parts* given that the first subset contains no key material is Lemma 52 (analz_keyfree_into_Un).

**Lemma 52.** *analz_keyfree_into_Un*

```
[|X ∈ analz (G ∪ H); G ⊆ keyfree|] ==> X ∈ parts G ∪ analz H
```

We prove Lemma 52 (analz_keyfree_into_Un) by first applying induction followed by the tactic *auto*, what will leave us with three sub goals. The first two goals concern the decryption of *Crypt K X* ∈ *parts G* which are easily provable by appealing to *parts.Body* in parts definition. The last sub goal regards *Crypt K X* ∈ *analz G* with *Key (invKey K)* ∈ *parts G*. Here we appeal to Lemma 51 (parts_keyfree) and to the fact that no *Key* is in *keyfree*, generated here by a *inductive_case* command in Isabelle/HOL.

### 5.3.3 General Validity Proofs

As our main goals shifted from proving correctness of the Franklin-Reiter protocol to the specification and investigations of suitability for a multicast event theory we present some of the proofs about the protocol we produced. We stress that these proofs are not the complete set we verified for the protocol. They are shown here to exemplify the suitability of the multicast event theory in dealing with knowledge distribution and with mixed environments of multicast and unicast. We focused on verifying the secrecy of the bids and that the declared winner participated in the bid casting as ways of showing that our multicast specification is usable and capable of representing real problems.

We will start by looking to Lemma 53 (bid_secrecy), which concern the secrecy of $v$ the coin's face value. Concomitantly we have other two lemmas in the similar form for $w$ and for $\{|v|\}_{Kr_{Bank}}$. This lemma states that if an event with syntax of the one yielded by rule *FR1* is in the trace of events, and the bidder $B$ is not colluding with the *Spy* and the *Spy* is not in the multicast group of auction servers, then coin's face value $v$ is not in the knowledge of the *Spy*.

**Lemma 53.** *bid_secrecy*

```
[| Multicast B multicast_group (λC. {|
      Nonce aid,
      Crypt (pubK C)({|Nonce (share (nat t, multicast_group, C)
                 {|Agent B, Nonce v, Nonce w|}),
                 Nonce aid|}),
      Nonce ( pub_share (nat t, multicast_group, C)
                 (signOnly (priSK Bank) (Nonce v))),
      Crypt (pubK C)(Nonce(priv_share (nat t, multicast_group, C)
                 (signOnly (priSK Bank)(Nonce v))))|}))∈ set evs)
B ∉ bad; Spy ∉ set multicast_group; evs ∈ fr |]
==> Nonce v ∉ analz (knows Spy evs)
```

Proving Lemma 53 (bid_secrecy) is a tough exercise as usual for secrecy properties. We start by using the usual proof method for secrecy lemmas by first preparing the induction and applying it, followed by the simplifier. We are left with seven sub goals, representing the two *Fake* rules and the five protocol steps. The Fake rule can be proven appealing to fact Fake_analz_eq[1] that states that $X \in$ *synth( analz H) ==> synth (analz (insert X H)) = synth (analz H)*. The sub goals regarding messages two to five are proven appealing to the fact that $v \notin$ *sessionIDs* and to the function congruence rule to eliminate the $\lambda$ expression in the multicast events. Proving the sub goal for message one involves applying the tactic *auto* augmented with the destruction rule for function congruence as usual for multicast messages. This yields seven new sub goals. The first one is proven resorting to the fact that $v \notin$ *sessionIDs*. Another two of these sub goals are proven appealing to *analz_into_parts* and *shares_shares* discussed in Chapter 4. This is the fact that all sharing primitives belong to the set *shares*. The next two sub goals are proven by appealing to *Multicast_imp_in_parts_spies*, shown in chapter 3 and the fact that if $v$ is inside a share it belongs to the set *used*. The final two sub goals are proven appealing to *analz_keyfree_into_Un* shown above and the fact that $v$ is not in sessionIDs.

Following to our next example, Lemma 54 (MSG5_imp_MSG1) states that if the unicast event *Says* yielded by rule *FR5* is in the trace of events declaring $B$ the winner of the auction, this implies that $B$ sent a *Multicast* event yielded by the rule *FR1* casting his bid to the auction servers in the multicast group running the auction *aid*. This lemma is shown to illustrate the ability our new multicast events theory has to integrate different message casting modes within the same protocol infrastructure.

**Lemma 54.** *MSG5_imp_MSG1*

```
[|Says S B {|Nonce aid, Agent B,
  sign (invKey (pubSK S)) {|Nonce aid, Agent B|}|} ∈ set evs;
  S ≠ Spy; evs ∈ fr|]
==> ∃XF multicast_group Bank v w.
    Multicast B multicast_group (λC. {|
      Nonce aid,
      Crypt (pubK C)({|Nonce (share (nat t, multicast_group, C)
                 {|Agent B, Nonce v, Nonce w|}),
                 Nonce aid|}),
      Nonce ( pub_share (nat t, multicast_group, C)
                 (signOnly (priSK Bank) (Nonce v))),
      Crypt (pubK C)(Nonce(priv_share (nat t, multicast_group, C)
                 (signOnly (priSK Bank)(Nonce v))))|})∈ set evs)
```

The proof of Lemma 54 (MSG5_imp_MSG1) involves the usual steps required to prepare and apply the induction, followed by the application of tactic *auto* augmented with the function congruence fact and the fact *bidsM* shown above.

Both lemmas shown here demonstrate the ability of integrating our multicast events theory into the verification of mixed environment protocols such as the Franklin-Reiter protocol. Especially the proofs such as the one for Lemma 53 (bid_secrecy) show that there is an increase in complexity in creating proofs for some goal such Secrecy. But this

[1]Not shown in this thesis

complexity is manageable and in general terms lets us follow the usual proof strategy as in unicast.

On the other hand, a lemma such as Lemma 54 (MSG5_imp_MSG1) shows us the flexibility of our specification by letting us reason about peers knowledge regardless of message casting framework we are using in a message. This example shows us that constructing a mixed environment proof has the same complexity as constructing a single framework one.

### 5.3.4 Interpreting of Findings

Although our verification about the Franklin-Reiter protocol was not thorough enough to verify all claims made by the authors, it enabled us to systematically study some of the constructions of the protocol and be able to informally find some design faults.

The first and main issue we detected with the protocol during our verification exercise was the lack of availability to prove goals using the bidder's point of view. The authors seemed to worry about embedding novel mechanisms and forgot to make available to all peers the information needed to build their conclusion over their minimal trust base. This creates an unbalanced protocol in the sense that the goals are available to only one subset of peers executing the protocol.

One clear example of this lack of balance in the protocol information flow lets us conclude that the losing bidders cannot assert that they lost in a fair way. As there is no message flowing back to the losing bidders informing them during the other phases of the auction protocol, there is no way to prove that they lost fairly, using the principle of Goal Availability.

Another point that our initial verification shows concerns the requirements of multicasts messages the protocol uses. The authors require their atomic multicast implementation to deliver authenticated messages. As this requirement is not a standard one in protocol design, we had to assume that the *Spy* was not colluding with sending peers. This makes our proof regarding Secrecy very weak taking into consideration the protocol's threat model.

## 5.4 Considerations

In this chapter we presented parts of our verification of the Franklin-Reiter sealed bid auction protocol with the focus of showing the suitability of our new multicast events theory being used in a mixed environment protocol. We first discussed the Franklin-Reiter protocol and outlined its properties and threat model. We then went through its assumptions and design, which include a scheme where mixed message frameworks are used. We also presented the author's proposal for protecting the bidder anonymity against the auction house. We concluded this section outlining some known issues of the protocol

We followed with the specification in Isabelle/HOL using the inductive method for the Franklin-Reiter protocol. We specified some auxiliary functions we designed to help us with a more concise specification. We specified the protocol using the Inductive method. Following the specification, we outlined the proofs for our auxiliary functions and showed two main lemmas we proved regarding protocol properties. The objectives of those proofs were to demonstrate the feasibility of using our new multicast events theory in the verification of a real protocol. We concluded with some of the outcomes regarding the protocol's properties we learned with this exercise.

Our main contribution in this chapter, aside the initial verification of the Franklin-Reiter protocol and showing the feasibility of our multicast events theory specification is the study and creation of methods to simplify the reasoning about the predicate *analz* in cases where no key material is involved.

— *Success has always been easy to measure. It is the distance between one's origins and one's final achievement.*

Michael Korda

# 6 Final Remarks

Security protocols have been an active area for computer security research in the last three decades. Together with their design strategies we had the necessity of understanding them. They create a puzzle of different security pieces tied together to yield more complex security properties. The need of understanding why some combinations of different techniques are able to achieve bigger and more complex goals while some others does not, lead to what we know today as protocol verification.

Informal verification tries to draw conclusions regarding security protocols based on the capacity of humans in learning from their mistakes, follow their intuition and executing empirical procedures. Although simple at a first glance, informal reasoning about security protocols properties is difficult to conduct in a thorough manner. As the goals and primitives become more complex, the verification process becomes unmanageable. The limitations of informality are the thoroughness of the verification process and the size of the protocols it can manage. These limitations are clear on their output, but nevertheless they provide a very good first line of approach even today.

Trying to address the increase in goal complexity and to unveil hidden features of informal reasoning, we saw the introduction of formal verification methods. Formal verification methods are approaches which deal with the problem of deciding whether a protocol fulfils its goals by applying mathematical proofs of correctness. The Inductive Method, the focus of this thesis, is a good example of formal verification procedure we can apply today. Its strong mathematical basis makes it powerful and yet flexible. To answer if a protocol holds the desired properties has been the aim of various formal methods for protocol verification. As we have seen in Chapter 2, these methods started as execution machines to represent protocols execution. Later we saw the introduction of logic formalisms tailored to address some of the most basic security goals. They introduced the use of mathematical foundations to the process of analysing the achievability of security properties by protocols. We also looked at the division of the protocol verification area including the two main strains of Model Checking and Theorem Proving. With the usage of generic tools or purpose built ones, the problem of protocol verification still remains undecidable. The management of undecidability is done by limiting the complexity of the verification process.

The Inductive Method was implemented over the Isabelle/HOL theorem prover, a tool that shows few limitations to the verification process while being built over a generic purpose system. This generality makes the inductive method a perfect test bed for novel formal verification efforts, especially for security protocols. It was already used to prove a series of classical protocols [99, 101] as well as some well-known industry grade protocols, such as the SET online payment protocol, Kerberos and SSL/TLS [26, 23]. One of its advantages is flexibility.

We also have seen the increased interest from protocol designers in the usage of novel security primitives and communication methods. The information revolution we live today requires the conversion of our everyday processes to their digital counterparts. This information revolution inherently means the introduction of more complex goals and the necessity of better exploring the infrastructure in place. From the protocol design point of view, we have seen the introduction of several new protocols that cannot be properly verified using the tools available today. As examples we can cite protocols that are based on communication methods other than Unicast and protocols that use novel and complex security primitives such as threshold cryptography. These new characteristics are present today in a varied set of new security protocols, such as Election Protocols [110] that make use of broadcasts and zero knowledge proofs, or even to the ongoing e-commerce revolution with protocols for sealed-bid auctions [58].

Our contributions in this thesis comes to address some of the shortcomings for the formal verification methods and in special from the Inductive Method. Our effort into extending the Inductive Method was to enable it to reason about non-Unicast message casting methods. Based on the assumption that other message casting methods are special cases for multicast, we built a theory for representing multicast communication (Chapter 3). The central idea of building such theory was that we should have a more flexible infrastructure for the inductive method to represent whole new classes of protocols. We also did experimentation with our proposed design to show its backward compatibility (Chapter 3) and its novel implementation capabilities (Chapter 5).

In the effort of expanding the inductive method to the verification of Byzantine security protocols (Chapter 5), we studied the introduction of threshold security primitives and the impact of these changes to the method. We implemented a basic formalism for presenting secret sharing schemes (Chapter 4). We also investigated the impact of embedding these primitives deeper within the method. To conclude, we used these primitives for a verification of secrecy in a Byzantine Security Protocol for sealed-bid auctions.

As remarks of our work with the Inductive Method and its extension, we can verify the veracity of some critics regarding the method. Its very steep learning curve and the expensive human efforts for constructing some verification are clear drawbacks for the method. In terms of learning curve, we detected that the time to train a security researcher to understand the method and being capable of producing initial results is on the range of 45 man-weeks. But, we also noticed that, the training for the Inductive Method can be made easier by first appealing to first-order logic versions of it. First-order versions of the inductive method help beginners to fix the concepts behind the inductive definitions and assertions while hiding the complexity of higher-order inductive proofs. During our learning process by using a first order method, we successfully discovered attacks in two real world protocols.

Nevertheless the effort in producing Isabelle/HOL proofs is big for a protocol and in average of ten weeks for a new protocol [20], the deep understanding of the protocols constructions becomes evident to the person performing the verification. Although we see a push in the security protocol verification community to button push tools to do such verifications, by our experience, we believe that for capturing novel attacks and to represent novel security protocol goals, we need powerful and yet extensible tools such as the Inductive Method and the theorem prover Isabelle/HOL. Once these novelty points were tested and understood, their embedding in the "push button" tool is a matter of time.

On the ongoing and future research of security protocol verification, we believe we will see more demand for methods capable of being extended and capable of representing the interesting subtleties of the ever new designs. Extensibility becomes a key issue to this field because of the stability it already assumed. No major theoretical breakthrough happened in the last decade. We see that the community will be working further in having more and more coverage for verification methods regarding the security protocols ever growing set.

On the aspects of our direct contributions to protocol verification, we envisage the verification of election protocols as being the next big step. With the setup for supporting Multicast and Broadcast it is possible for a whole new family of such protocols to be verified by the Inductive Method. This will incur in the investigation of new security goals such as Anonymity which is very much related to the notion of privacy such voting protocols require.

We also must recall our effort in extending the cryptographic primitives. This study may enable the inclusion of new primitives such as zero-knowledge proofs, what can advance even more the verification of such novel election protocols. This area of cryptographic primitives extensibility and coverage seems not very explored up to this date by most methods.

Concluding this manuscript, we see that the extension of verification tools for evaluating the security of protocols not based on Unicast will demand the revision of threat models. We have seen lately some exercises into new abstract models for threats evolving towards more real and asymmetric scenarios. In our experimentation with the non-Unicast events, we have seen that knowledge is distributed to peers in a different ways as in Unicast. This creates asymmetry of knowledge distribution, meaning for example that one peer is able to infer other peers knowledge by knowing they belong or not to the multicast group. It seems that scenarios where counter attacks are possible or avoidable can become more important in the decision of the achievability for some security goals.

# Bibliography

[1] Abadi and Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15, 2002.

[2] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 10 January 1999.

[3] J. Abley and K. Lindqvist. Operation of Anycast Services. Technical Report 4786, Internet Engineering Task Force, December 2006.

[4] Z. Albanna, K. Almeroth, D. Meyer, and M. Schipper. IANA Guidelines for IPv4 Multicast Address Assignments. RFC 3171, Internet Engineering Task Force, August 2001.

[5] Giuseppe Anastasi, Alberto Bartoli, Nicoletta De Francesco, and Antonella Santone. Efficient verification of a multicast protocol for mobile computing. *Comput. J*, 44(1):21–30, 2001.

[6] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* Wiley, 1 edition, January 2009.

[7] Myla Archer. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME*. In *Workshop on Issues in the Theory of Security*, Portland, OR, 2002.

[8] Alessandro Armando, David Basin , Yohann Boichut, Yannick Chevalier, and al Et. The AVISPA tool for the automated validation of internet security protocols and applications. In Etessami Kousha and Rajamani Sriram, editors, *Computer-Aided Verification , Edinburgh, Scotland, UK, 06/07/05-10/07/05*, pages 1–5. Springer-Verlag, 2005.

[9] Alessandro Armando and Luca Compagna. SATMC: A SAT-based model checker for security protocols. In José Júlio Alferes and João Alexandre Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 730–733. Springer, 2004.

[10] Wihem Arsac, Giampaolo Bella, Xavier Chantry, and Luca Compagna. Validating security protocols under the general attacker. In Pierpaolo Degano and Luca Viganò, editors, *ARSPA-WITS*, volume 5511 of *Lecture Notes in Computer Science*, pages 34–51. Springer, 2009.

[11] Wihem Arsac, Giampaolo Bella, Xavier Chantry, and Luca Compagna. Multi-attacker protocol validation. *Journal of Automated Reasoning*, 45:1–36, 2010.

[12] Jian Yin Arun, Jian Yin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Byzantine fault-tolerant confidentiality. In *International Workshop on Future Directions in Distributed Computing*, pages 12–15, 2002.

[13] Ashar Aziz and Whitfield Diffie. Privacy and authentication for wireless local area networks. *IEEE Personnal Comm*, 1(1):25–31, July 1993.

[14] Michael Backes and Birgit Pfitzmann. Relating symbolic and cryptographic secrecy. In *IEEE Symposium on Security and Privacy*, pages 171–182. IEEE Computer Society, 2005.

[15] David Basin, Sebastian Möersheim, and Luca Viganò. An on-the-fly model-checker for security protocol analysis. In *Eight ESORICS*, volume 2808 of *Lecture Notes in Computer Science*, pages 253–270. Springer-Verlag, Berlin Germany, 2003.

[16] David A. Basin. Lazy infinite-state analysis of security protocols. In Rainer Baumgart, editor, *CQRE*, volume 1740 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 1999.

[17] David A. Basin, Sebastian Mödersheim, and Luca Viganò. OFMC: A symbolic model checker for security protocols. *Int. J. Inf. Sec*, 4(3):181–208, 2005.

[18] Giampaolo Bella. *Inductive Verification of Cryptographic Protocols*. PhD thesis, University of Cambridge, March 2000.

[19] Giampaolo Bella. Inductive verification of smart card protocols. *Journal of Computer Security*, 11(1):87–132, 2003.

[20] Giampaolo Bella. *Formal Correctness of Security Protocols*. Information Security and Cryptography. Springer, 2007.

[21] Giampaolo Bella, Stefano Bistarelli, and Fabio Massacci. Retaliation: Can we live with flaws? In *Workshop on Information Security Assurance and Security*, 2005.

[22] Giampaolo Bella, Cristiano Longo, and Lawrence C. Paulson. Is the verification problem for cryptographic protocols solved? In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols Workshop*, volume 3364 of *Lecture Notes in Computer Science*, pages 183–189. Springer, 2003.

[23] Giampaolo Bella and Lawrence C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. *Lecture Notes in Computer Science*, 1485, 1998.

[24] Giampaolo Bella and Lawrence C. Paulson. Mechanising BAN Kerberos by the inductive method. *Lecture Notes in Computer Science*, 1427, 1998.

[25] Giampaolo Bella and Lawrence C. Paulson. Accountability protocols: Formalized and verified. *ACM Trans. Inf. Syst. Secur.*, 9(2):138–161, 2006.

[26] Giampaolo Bella, Lawrence C. Paulson, and Fabio Massacci. The verification of an industrial payment protocol: the set purchase phase. In *9th ACM conference on Computer and communications security*, pages 12–20, New York, NY, USA, 2002. ACM Press.

[27] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer-Verlag, Berlin Germany, 1994.

[28] Josh Cohen Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In *8th Annual International Cryptology Conference on Advances in Cryptology*, pages 27–35, London, UK, 1990. Springer.

[29] Bob Blakley, G. R. Blakley, A. H. Chan, and J. L. Massey. Threshold schemes with disenrollment. In Ernest F. Brickell, editor, *Advances in Cryptology—CRYPTO '92*, volume 740 of *Lecture Notes in Computer Science*, pages 540–548. Springer-Verlag, 1993, 16–20 August 1992.

[30] George R. Blakley. Safeguarding cryptographic keys. In *National Computer Conference*, pages 313–317. AFIPS, 1979.

[31] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop*, pages 82–96, Washington - Brussels - Tokyo, June 2001. IEEE.

[32] Carlo Blundo, Alfredo De Santis, Ugo Vaccaro, Amir Herzberg, Shay Kutten, and Moti Yong. Perfectly secure key distribution for dynamic conferences. *Inf. Comput.*, 146(1):1–23, 1998.

[33] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[34] Yohan Boichut, Pierre-Cyrille Héam, and Olga Kouchnarenko. Automatic verification of security protocols using approximations. INRIA Technical Report, 2005.

[35] Steve Brackin. A HOL extension of GNY for automatically analyzing cryptographic protocols. In *9th IEEE Computer Security Foundations Workshop*, pages 62–77, Washington - Brussels - Tokyo, June 1996. IEEE.

[36] Ernest F. Brickell and Daniel M. Davenport. On the classification of ideal secret sharing schemes (extended abstract). In G. Brassard, editor, *Advances in Cryptology*, volume 435 of *Lecture Notes in Computer Science*, pages 278–285. Springer-Verlag, 1990, 1989.

[37] Ian Brown, Colin Perkins, and Jon Crowcroft. Watercasting: Distributed watermarking of multicast media. In Luigi Rizzo and Serge Fdida, editors, *Networked Group Communication*, volume 1736 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 1999.

[38] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[39] Franco Callegati, Walter Cerroni, and Marco Ramilli. Man-in-the-middle attack to the https protocol. *IEEE Security and Privacy*, 7:78–81, 2009.

[40] Ran Canetti, Juan A. Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. Multicast security: A taxonomy and some efficient constructions. In *INFOCOM*, pages 708–716, 1999.

[41] Dario Catalano and Rosario Gennaro. New efficient and secure protocols for verifiable signature sharing and other applications. *Journal of Computer and System Sciences*, 61, 2000.

[42] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Comm. of the ACM*, 28(10), October 1985.

[43] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash (extended abstract). In S. Goldwasser, editor, *Advances in Cryptology*, volume 403 of *Lecture Notes in Computer Science*, pages 319–327. Springer-Verlag, 1990, 1988.

[44] Yannick Chevalier and Laurent Vigneron. Automated unbounded verification of security protocols. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 324–337. Springer-Verlag, July 27–31 2002.

[45] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In *26th Annual Symposium on Foundations of Computer Science*, pages 383–395, Portland, Oregon, 21–23 October 1985. IEEE.

[46] Ernie Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[47] Miguel Correia, Lau Cheuk Lung, Nuno Ferreira Neves, and Paulo Verissimo. Efficient byzantine-resilient reliable multicast on a hybrid failure model. In *Symposium on Reliable Distributed Systems*, pages 2–11, Osaka, Japan, October 2002. IEEE.

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

[49] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36:372–421, December 2004.

[50] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Comm. ACM*, 24(7):533–536, August 1981.

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

[52] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, July 1985.

[53] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct. In *Conference on Security and Privacy*, pages 160–171. IEEE Press, May 1998.

[54] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Symposium on Foundations of Computer Science (FOCS)*, pages 427–437. IEEE Computer Society Press, 1987.

[55] FIPS. *Advanced Encryption Standard (AES).* National Institute for Standards and Technology, pub-NIST:adr, November 2001.

[56] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCann, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *Networking, IEEE/ACM Transactions on*, 5(6):784 –803, December 1997.

[57] Matthew K. Franklin and Michael K. Reiter. Verifiable signature sharing. In *Advances in Cryptology*, 1995.

[58] Matthew K. Franklin and Michael K. Reiter. The design and implementation of a secure auction service. *IEEE Transactions on Software Engineering*, 22(5):302–312, 1996.

[59] Felix C. Gärtner. Byzantine failures and security: Arbitrary is not (always) random. Technical report, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, 2003.

[60] Rosario Gennaro and Pankaj Rohatgi. How to sign digital streams. *Inf. Comput.*, 165(1):100–116, 2001.

[61] Shafi Goldwasser. Multi party computations: past and present. In *sixteenth annual ACM symposium on Principles of distributed computing*, pages 1–6, New York, NY, USA, 1997. ACM.

[62] Dieter Gollmann. On the verification of cryptographic protocols - A tale of two committees. *Electronic Notes in Theoretical Computer Science*, 32, 2000.

[63] Li Gong. A security risk of depending on synchronized clocks. *Operating Systems Review*, 26(1):49–53, 1992.

[64] Roberto Gorrieri, Fabio Martinelli, and Marinella Petrocchi. Formal models and analysis of secure multicast in wired and wireless networks. *J. Autom. Reasoning*, 41(3-4):325–364, 2008.

[65] T. Hardjono and B. Weis. The Multicast Group Security Architecture. RFC 3740 (Informational), March 2004.

[66] H. Harney and C. Muckenhirn. RFC 2094: Group key management protocol (GKMP) architecture, July 1997. Status: EXPERIMENTAL.

[67] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing, or: How to cope with perpetual leakage. In Don Coppersmith, editor, *Advances in Cryptology*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352. Springer-Verlag, Berlin Germany, 1995.

[68] Charles A. R. Hoare. *Communicating sequential processes.* Prentice-Hall International, Englewood Cliffs, N.J, 1985.

[69] Dijiang Huang and Deep Medhi. A byzantine resilient multi-path key establishment scheme and its robustness analysis for sensor networks. In *19th International Parallel and Distributed Processing Symposium*, Denver, CO, USA, April 2005.

[70] Sorin Iftene. Secret Sharing Schemes with Applications in Security Protocols. Technical Report TR 07-01, "Al.I.Cuza" University of Iaşi, Faculty of Computer Science, 2007.

[71] Tomasz Imielinski and Jorge Navas. GPS-Based Addressing and Routing. Technical Report 2009, IETF, November 1996.

[72] ISO/IEC. Iso/iec 9798-1:2010 information technology – security techniques – entity authentication – part 1: General. Technical report, International Organization for Standardization, Geneva, Switzerland., 2010.

[73] Masahiko Ito, Akinori Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. In *IEEE Globecom*, pages 99–102, 1987.

[74] Daniel Jackson. Automating first-order relational logic. In David S. Rosenblum, editor, *8th International Symposium on the Foundations of Software Engineering*, volume 25, 6 of *ACM Software Engineering Notes*, pages 130–139, NY, November 8–10 2000. ACM Press.

[75] Jay A. Kreibich. The mbone: the internet's other backbone. *Crossroads*, 2(1):5–7, 1995.

[76] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4, 1982.

[77] Gavin Lowe. An attack on the needham-schroeder public key authentication protocol. *Information Processing Letters*, 56(3):131–136, November 1995.

[78] Gavin Lowe. A hierarchy of authentication specifications. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 31 –43, June 1997.

[79] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[80] Wenbo Mao and Colin Boyd. Towards formal analysis of security protocols. In *Proceedings of the Computer Security Foundations Workshop*, pages 147–158, Washington - Brussels - Tokyo, June 1993. IEEE.

[81] Keith M Martin, Josef Pieprzyk, Rei Safavi-Naini, and Huaxiong Wang. Changing thresholds in the absence of secure channels. *Australian Computer Journal*, 31(2):34–43, 1999.

[82] Jean E. Martina and Luiz A. C. Boal. A formal analysis of the brazilian electronic bill of sale protocols. In *Proceedings of the Brazilian Symposium on Information and Computer System Security*, Gramado, Brazil, Semptember 2008. Brazilian Computer Society.

[83] Jean E. Martina, Tulio C. S. de Souza, and Ricardo F. Custódio. Openhsm: An open key life cycle protocol for public key infrastructure's hardware security modules. In *EuroPKI'07*, LNCS. Springer-Verlag, 2007.

[84] Nicholas F. Maxemchuk and David H. Shur. An internet multicast system for the stock market. *Transactions on Comp. Systems*, 19:2001, 2001.

[85] Robert J. McEliece and Dilip V. Sarwate. On sharing secrets and Reed-Solomon codes. *Communications of the ACM*, 24(9):583–584, September 1981.

[86] Catherine Meadows. Formal verification of cryptographic protocols: A survey. In *Proceedings of Asiacrypt 96*, 1996.

[87] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, February 1996.

[88] Catherine Meadows. Extending formal cryptographic protocol analysis techniques for group protocols and low-level cryptographic primitives. In Pierpaolo Degano, editor, *Workshop on Issues in the Theory of Security*, University of Geneva, Switzerland, July 2000.

[89] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput*, 204(10):1575–1596, 2006.

[90] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[91] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using mur-phi. In *IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society, 1997.

[92] National Bureau of Standards. *FIPS Publication 46-1: Data Encryption Standard*, January 1988.

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

[94] Roger M. Needham and Michael D. Schroeder. Authentication revisited. *ACM Operating Systems Review*, 21(1):7, January 1987.

[95] Dan M. Nessett. A critique of the Burrows, Abadi and Needham logic. *ACM Operating Systems Review*, 24(2):35–38, 1990.

[96] Barry Clifford Neuman and Stuart G. Stubblebine. A note in the use of timestamps as nonces. *Operating Systems Review*, 1993.

[97] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.

[98] Dave Otway and Owen Rees. Efficient and timely mutual authentication. *ACM Operating Systems Review*, 21(1):8–10, January 1987.

[99] Lawrence C. Paulson. Mechanized proofs for a recursive authentication protocol. In *Proceedings of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[100] Lawrence C. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Technical Report 413, University of Cambridge, Computer Laboratory, January 1997.

[101] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[102] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer-Verlag, 1992, 1991.

[103] Antonio Pinto and Manuel Ricardo. Smiz - secure multicast iptv with efficient support for video channel zapping. In *Proceedings of the NAEC 2008, Networking and Electronic Commerce Research Conference 2008*, Lake Garda, Italy, September 2008.

[104] Projeto NF-e. Manual de integra cão do contribuinte - padrões técnicos de comunica cão. Technical Report 2.0.2, ENCAT, Junho 2007.

[105] B. Quinn and K. Almeroth. IP Multicast Applications: Challenges and Solutions. RFC 3170 (Informational), September 2001.

[106] Ron L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[107] Andrew W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *CSFW*, pages 98–107. IEEE CS, 1995.

[108] Andrew W. Roscoe. Intensional specifications of security protocols. In *9th IEEE Computer Security Foundations Workshop*, pages 28–38, Kenmare, Co. Kerry, Ireland, June 1996. IEEE Computer Society Press.

[109] Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with a finite number of sessions and composed keys is $NP$-complete. *Theor. Comput. Sci.*, 299(1-3):451–475, 2003.

[110] Peter Y. A. Ryan, David Bismark, James Heather, Steve Schneider, and Zhe Xia. Prêt à voter: a voter-verifiable voting system. *IEEE Transactions on Information Forensics and Security*, 4(4):662–673, 2009.

[111] Peter Y. A. Ryan and Steve A. Schneider. An attack on a recursive authentication protocol — A cautionary tale. *Information Processing Letters*, 65(1):7–10, January 1998.

[112] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In Michael Wiener, editor, *Advances in Cryptology*, volume 1666 of *Lecture Notes in Computer Science*, pages 148–164. Springer-Verlag, Berlin Germany, 1999.

[113] Johann Schumann. Automatic verification of cryptographic protocols with SETHEO. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 87–100, Berlin, July 13–17 1997. Springer.

[114] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[115] Gustavus J. Simmons. The subliminal channel and digital signatures. In *Advances in Cryptology*, volume 209 of *Lecture Notes in Computer Science*. Springer Verlag, 1984.

[116] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.

[117] Markus Stadler. Publicly verifiable secret sharing. In Ueli Maurer, editor, *Advances in Cryptology*, volume 1070 of *Lecture Notes in Computer Science*, pages 190–199. Springer-Verlag, Berlin Germany, 1996.

[118] Graham Steel and Alan Bundy. Attacking group multicast key management protocols using coral. *Electr. Notes Theor. Comput. Sci*, 125(1):125–144, 2005.

[119] Graham Steel and Alan Bundy. Attacking group protocols by refuting incorrect inductive conjectures. *J. Autom. Reason.*, 36(1-2):149–176, 2006.

[120] Graham Steel, Alan Bundy, and Ewen Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. *The AISB Journal*, 1(2):169–182, 2002.

[121] Paul Syverson, Catherine Meadows, and Iliano Cervesato. Dolev-yao is no better than machiavelli. In Pierpaolo Degano, editor, *Workshop on Issues in the Theory of Security*, University of Geneva, Switzerland, July 2000.

[122] Mana Taghdiri and Daniel Jackson. A lightweight formal analysis of a multicast key management scheme. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *FORTE*, volume 2767 of *Lecture Notes in Computer Science*, pages 240–256. Springer, 2003.

[123] Martin Tompa and Heather Woll. How to share a secret with cheaters. *J. Cryptology*, 1(2):133–138, 1988.

[124] Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *16th International Conference on Automated Deduction*, pages 314–328, London, UK, 1999. Springer-Verlag.

[125] Chung Kei Wong and Simon S. Lam. Digital signatures for flows and multicasts. In *IEEE/ACM Transactions on Networking*, pages 502–513, 1998.

[126] Kok Meng Yew, M. Zahidur Rahman, and Sai Peck Lee. Formal verification of secret sharing protocol using coq. In *5th Asian Computing Science Conference on Advances in Computing Science*, pages 381–382, London, UK, 1999. Springer-Verlag.

[127] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Coca: A secure distributed online certification authority. *ACM Trans. Comput. Syst.*, 20(4):329–368, 2002.