On the Formal Verification of an Intrusion-Tolerant Group Communication Protocol

Mohamed Layouni

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montreal, Quebec, Canada

August 2003

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisisitons et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-83868-4
Our file Notre référence
ISBN: 0-612-83868-4

ABSTRACT

On the Formal Verification of an Intrusion-Tolerant Group Communication Protocol

Mohamed Layouni

Intrusion-tolerance is the technique of using fault-tolerance to achieve security properties. Assuming faults to be unavoidable, the main goal of intrusion-tolerance is to preserve an acceptable, though possibly degraded, service of the overall system despite intrusions at some of its parts. In this thesis, we are concerned with the formal specification and verification of the Enclaves protocol: an intrusion-tolerant platform for secure group communication. We formally specify the three modules of Enclaves, namely, Authentication, Leaders Agreement and Group Key Management. Then we derive a correctness proof for the whole protocol using an adaptive combination of techniques, namely, model checking, theorem proving and mathematical analysis. We use the Murphi model checking tool to verify *authentication*, then the PVS theorem prover to formally specify and prove proper *Byzantine agreement*, agreement *termination* and *integrity*, and finally we analytically prove *robustness* and *unpredictability* of the Group Key Management module.

To My Dear Parents...

ACKNOWLEDGEMENTS

First, I would like to express my heartfelt gratitude to Dr. Sofiène Tahar, my thesis supervisor, for his most helpful guidance, support and for always pushing me to do better. I do thank him also for providing me with priceless opportunities and for generously financing my research travels. I do owe him credits for that.

I would like also to thank Dr. Jozef Hooman, of University of Nijmegen, The Netherlands, for his constant help with the PVS proofs. Collaborating with him has been such a worthy experience.

I would extend thanks also to the thesis examination committee members for providing me with constructive feedback to improve on the presentation and contents of this thesis.

Finally, I would like to express special thanks to my colleagues of the Hardware Verification Group (HVG) at Concordia University; working with them has been, for me, one of the most rewarding experiences.

This work has been supported in part by a studentship from the Universitary Mission of Tunisia in North America.

Mohamed Layouni

Montreal, Canada, August 2003

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ACRONYMS xi

1 Introduction **1**

1.1 Security Protocols 1

1.2 Formal Analysis of Security Protocols 4

1.2.1 Formal Specification 5

1.2.2 Verification Techniques 6

1.3 Scope and Contributions of the Thesis 10

1.4 Related Work 13

1.5 Outline of the Thesis 16

2 Intrusion-Tolerant Enclaves **18**

2.1 Introduction 18

2.2 Overview on Enclaves 20

2.2.1 Enclaves Services 20

2.2.2 Centralized Architecture 22

2.2.3 Intrusion-Tolerant Architecture 23

2.2.4 Protocol Execution 25

2.3 Enclaves Protocols 27

2.3.1 Authentication 27

2.3.2 Leader Coordination 28

2.3.3 Group-Key Management 30

3 Proving Authentication by Model Checking in Murphi 36

3.1 Protocol Specification in Murphi 37

3.1.1 Modeling Protocols 38

3.1.2 Specifying Properties 38

3.2 Local Authentication in Enclaves: System Model 39

3.2.1 Modeling Users and Leaders 40

3.2.2 Modeling Intruders 44

3.2.3 Properties Specification 45

3.3 Experimental Results 46

3.4 Concluding Remarks 48

4 Proving Byzantine Agreement by Theorem Proving in PVS 51

4.1 Modeling Byzantine Agreement in PVS 53

4.1.1 Timed Automata 53

4.1.2 Modeling Leaders Actions 54

4.1.3 Modeling States 56

4.1.4 Preconditions and Effects 58

4.1.5 Protocol Runs and Fault Assumptions 60

4.2 Correctness Proofs 61

4.3 Concluding Remarks 64

5 Proving Group-Key Management by Mathematical Analysis 66

5.1 Enclaves Group-Key Management: A Verifiable Secret Sharing Scheme 67

5.2 Provable Security : Approaches and Models 71

5.2.1 Concrete vs. Asymptotic Approaches 71

5.2.2 Random Oracle vs. Standard Models 72

5.3 Security Analysis of the Group-Key Management Module 74

5.4 Concluding Remarks 76

6 Conclusion and Future Work 78

Bibliography 81

LIST OF TABLES

3.1 Model Checking Experimental Results 47

LIST OF FIGURES

2.1 Centralized Architecture 22

2.2 Intrusion-Tolerant Architecture 23

2.3 Protocol Execution 26

LIST OF ACRONYMS

BAN	Burrows Abadi and Needham' (Belief Logic)
HOL	Higher Order Logic (Theorem Prover)
IDS	Intrusion Detection System
ISO	International Standardization Organization
NRL	Naval Research Lab (Protocol Analyzer)
OBDD	Ordered Binary Decision Diagrams
PVS	Prototype Verification System
ROM	Random Oracle Model
RSA	Rivest, Shamir and Adleman (Public-Key Encryption Scheme)
SPC	Security Protocol Calculus
TAME	Timed Automata Modeling Environment
TESLA	Timed Efficient Stream Loss-tolerant Authentication

Chapter 1

Introduction

1.1 Security Protocols

Research in network security has been always, and continue to be, a big challenge to the scientific community as well as governments and corporations alike. The main goal of network security is to prevent intruders from performing any malicious activity directed towards computer systems or any services they provide. Security protocols come in two main flavors: those using symmetric key encryption (e.g., [76, 77]) and those based on public key cryptography (e.g., [93]). We distinguish also between those relying on (one or more) trusted third parties to carry out some agreed function and those that operate purely between two communicating principals that wish to achieve some mode of authentication [64]. There are further distinctions that can be made: the number of messages involved in the protocols (e.g. one-pass, two-pass, three-pass, etc.) and whether one principal wishes to convince the second of some matter (one-way or unilateral authentication, e.g., [53]) or whether both parties wish to convince each other of something (two-way or mutual authentication, e.g., [78]). These distinctions are also made by the ISO entity authentication standards [55]. Lately, and in an attempt to better understand security protocols, researchers have adopted a number of generic properties such as confidentiality, authentication and non-repudiation to characterize systems security (e.g., [8, 42, 43, 103]).

Despite the variety of designs and the numerous efforts seen in the literature, It is well acknowledged that any sufficiently complex computer system will still have vulnerabilities. In other words, security flaws can be possibly transformed, but cannot be totally removed and will always remain inherent to computer systems (e.g., authentication protocols solve the *entity authentication* problem assuming the *key distribution* problem solved, which in reality is very hard to solve). In an attempt to reinforce systems security, researchers focused their attention on the concept of intrusion detection and came up with a variety of theories and tools in that direction (e.g., [28, 80, 97]). This field was believed to be very promising until the MIT Lincoln Labs publish a series of quite exhaustive evaluation studies [73] covering a number of well-respected research and commercial intrusion detection systems (IDS), e.g., [22, 54, 85]. Two aspects of the results are very intriguing. First, new and novel attacks present a formidable challenge to these systems. Second, little improvement in performance (e.g., true detection rate and false positive rate) was shown by those systems after years of further development [48].

It is widely agreed [47, 48, 80] that the serious limitation of "intrusion detection" is due to the fact that, as soon as we focus our attention on the attacks themselves, we cannot expect to develop a general protection mechanism because all attacks are not well-defined and there are always unknown attacks.

The alternative approach to the problem is intrusion tolerance. Assuming that vulnerabilities are inherently unavoidable, the main goal of intrusion tolerance is to maintain an acceptable, though possibly degraded, level of service regardless of intrusions at some of the system components. A number of intrusion tolerant protocols have been developed during the last decade (e.g., [5, 90, 92]). They make extensive use of basic fault-tolerance techniques (e.g., redundancy and diversity) and apply them to achieve security goals. The results (in terms of protocol robustness) have been very promising. One big challenge to intrusion tolerance, however, is to answer the following question: How to concretely and rigorously model compromised (possibly dishonest) components which may exhibit a very unpredictable behavior. One of the primer techniques for the validation of intrusion tolerant systems is the use of formal methods.

1.2 Formal Analysis of Security Protocols

Many formal techniques and tools have been developed to reason about security and fault-tolerant protocols. These techniques provide a rigorous mathematical framework for the description and analysis of any protocol. According to Meadows [70] taxonomy, formal methods used in security fall into four categories:

Type I

These are approaches that model and verify a protocol using specification languages and verification tools not specifically developed for the analysis of cryptographic protocols or fault-tolerant protocols. These include Analytical Mathematics (for provable security) (e.g., [88]), Theorem Proving (e.g., [81]) and Model Checking (e.g., [31]).

Type II

In these approaches, a protocol designer develops expert systems to create and examine different scenarios, from which he/she may draw conclusions about the security of the protocols being studied. Example expert systems are Millen's Interrogator [72] and the NRL Analyzer [71].

Type III

These approaches model the requirements of a protocol family using logics developed specifically for the analysis of knowledge and belief. Major research has been done about techniques of this type (e.g., [1, 3, 16, 45, 101]).

Type IV

Finally, these approaches develop a formal model based on the algebraic term-rewriting properties of cryptographic systems (e.g., [32, 99]).

Our work fits within the class of techniques of type I. Our choice has been mainly motivated by the generality of these techniques and also by the maturity reached by many tools implementing them. In the following, we briefly outline the mainstream specification and verification techniques used, as well as their application domains.

1.2.1 Formal Specification

Specification is the process of describing a system and its desired properties. Formal specification uses a language with a mathematically-defined syntax and semantics. The target system properties might include functional behavior, timing behavior, performance characteristics, etc. So far, formal specification has been most successful for behavioral properties, real-time constraints, security protocols, and architectural designs. One current trend is to find a way to formally express impossibility results such as the Diffie-Hellman [30] and the Discrete logarithm [67] problems. These are usually fundamental correctness arguments for a large number of cryptographic schemes.

1.2.2 Verification Techniques

Model Checking

Model checking [27] is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. Roughly speaking the check is performed as an exhaustive state space search which is guaranteed to terminate since the model is finite. The main challenge in model checking is in devising algorithms and data structures that allow the tools to handle large search spaces. Two general approaches to model checking are used: Explicit state enumeration and Symbolic model checking. In the first, the states of the system under verification are basically encoded and stored in a table and then checked against the desired properties. Many state reduction techniques have been developed to help reduce the state space without affecting the ability of the tool to discover bugs (e.g., [26, 35, 56]). Other techniques, called algorithmic techniques have been devised in order to optimize, both, the search and storage procedures without reducing the size of the state space (e.g., [98, 102]). Both techniques have proved to be useful in many notable verification examples (e.g., [56, 58, 100]). The second approach to model checking is Symbolic Model Checking [69]. This approach brought some remedy to the state space explosion problem of model checking and is based on the use of Ordered Binary Decision Diagrams (OBDDs) [15]. Using OBDDs allows for an efficient representation of systems state transitions, thereby increasing the size of systems that could be verified.

To date, the practice of model checking succeeded to handle systems with over 10^{120} reachable states. It is also believed that, by using appropriate abstraction techniques, model checkers can support systems with an essentially unlimited number of states [27]. In a nutshell, the discipline proved powerful "enough" that it is becoming a widely used tool in the verification of newly developed industrial designs.

Theorem Proving

Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. The logic is given by a formal system based on a set of axioms and inference rules. Theorem proving is the process of finding proofs using axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas. Nowadays, theorem provers are being increasingly used in the mechanical verification of safety critical properties of hardware and software systems.

Theorem provers based on higher-order logic are distinguished by a high level of expressiveness and generality compared to model checkers. They do, however, require a lot of user expertise and interactivity. A significant amount of work has been witnessed in an effort to automate these tools. One notable example is the Timed Automata Modeling Environment (TAME) [7] developed at the Center for High Assurance Computer Systems of the Naval Research Laboratory (NRL). TAME is an interface to the theorem proving system PVS [81], it provides a set of templates for specifying timed automata, a number of auxiliary theories, and a set of specialized PVS strategies to reason about them. In contrast to model checkers, theorem provers can deal with infinite state spaces. They rely on techniques like structural induction to prove over infinite domains. Although, theorem proving is a slow process that requires much of human interaction, it gives invaluable insight to the user on the system or the property being proved.

Mathematical Analysis: Provable Security

What is provable security? In fact, provable security stems from the concept of *reductions* in complexity theory. A reduction is a common way to solve a new problem in complexity theory. The paradigm is as follows. Take some goal, like achieving privacy via encryption. The first step is to make a formal adversarial model and define what does it mean for an encryption scheme to be secure. With this in hand, a particular scheme, based on some particular atomic primitives, can be analyzed from the point of view of meeting a certain definition of security. Eventually, one shows that the scheme "works" via a *reduction* (e.g., [11, 68]). The reduction shows that the only way to defeat the protocol is to break the underlying atomic primitives. In other words, there is no need to directly cryptanalyze the protocol, i.e., if one is able to find a weakness in the protocol then necessarily he or she would have exploited a weakness in the underlying atomic primitives. Therefore, it is more profitable (in terms of complexity) and sufficient to focus on the atomic primitives. And if we believe the latter are secure, we know, without further cryptanalysis of the protocol, that the protocol is secure.

In order to enable a reduction one must also have a formal notion of what is meant by the security of the underlying atomic primitives: what attacks, exactly, does it withstand? For example, we might assume RSA (Rivest, Shamir and Adleman public-key encryption algorithm [91]) is a collision-resistant one-way function, and consider a protocol P using RSA. A reduction based on the one-wayness of RSA to the security of P, is a transformation T with the following properties. Suppose one claims to be able to break protocol P. Let A_1 be the attack in question. The transformation T, takes A_1 and rewrites it, resulting in an attack A_0. This attack provably breaks RSA, but since we believe that RSA is unbreakable, then there could be no such attack on the protocol P. In other words, the protocol P is secure. Cryptographic reductions represent the backbone technique of provable security, and have been extended from classical to quantum cryptography as well [10].

1.3 Scope and Contributions of the Thesis

During the last decade, we have witnessed a substantial progress in the formal verification of cryptographic protocols; a wide variety of techniques has been developed to verify a number of key security properties ranging from *confidentiality* and *authentication* to *atomic transactions* and *non-repudiation* (e.g., [84, 94]). Nevertheless, most of the focus was either on two-party protocols (i.e., involving only a pair of users) or, in the best cases, on group protocols with centralized leadership (i.e., a presumably trusted fault-free server managing a group of users).

The work in this thesis is an attempt to the formal analysis of a more challenging and increasingly popular class of protocols, referred to as "Byzantine[1] fault-tolerant" protocols. In particular, we are considering the formal specification and verification of the Enclaves [33] protocol. Enclaves is a group-membership protocol with a distributed leadership architecture, where the authority of the traditional single server is shared among a set of n independent elementary servers, of which at most f could fail at the same time. The protocol has a maximum resilience of one third (i.e., $f \leq \lfloor \frac{n-1}{3} \rfloor$) and uses an algorithm similar to the consistent broadcast of Bracha and Toueg [13].

The primary goal of Enclaves is to preserve an acceptable group-membership service of the overall system despite intrusions at some of its sub-parts. For instance,

[1] A process is said to be a "Byzantine fault" or to exhibit a Byzantine behavior with respect to a given protocol if it does not follow the protocol rules and behaves arbitrarily [61].

an authorized user u who requests to join an active group of users should be eventually accepted, despite the fact that faulty leaders may coordinate their messages in such a way as to mislead non-faulty leaders (the majority) into disagreement, and thus into rejecting user u. Moreover, in order to prevent malicious leaders from leaking sensitive information (e.g., group keys) or providing clients with fake group keys, Enclaves uses a verifiably secure secret sharing scheme.

To achieve its intrusion-tolerant capabilities, Enclaves relies on the combination of a cryptographic authentication protocol, a Byzantine fault-tolerant leader agreement protocol and a secret sharing scheme. Although we assume the underlying cryptographic primitives and fault-tolerant components to be perfect, one cannot easily guarantee security of the whole protocol. In fact, several protocols had been long thought to be secure until a simple attack was found (see [25] for a survey). Therefore, the question of whether or not a protocol actually achieves its security goals becomes paramount.

An important issue that arises in formal verification of Byzantine fault-tolerant protocols, is the modeling of Byzantine behavior. How much power should be given to a Byzantine fault and how general should the model be to capture the arbitrary nature of a Byzantine fault behavior? These questions have been extensively studied [20, 60, 62] and continue to be a center of focus. In this thesis, we specify our faults to be able to crash and misbehave arbitrarily at any time. Faults are however limited by the protocol resiliency (no failure of more than the third of all principles) and by cryptographic constraints. More details about our fault assumptions are discussed in Chapter 4 of this thesis.

In our verification of Enclaves, we have adopted an adaptive combination of techniques chosen according to the nature of the correctness arguments in each module, the environment assumptions and the easiness of performing verification. For instance, we found it more profitable to model-check the authentication module by taking advantage of the reduction techniques available in the Murphi model checking tool [31]. The Byzantine leaders agreement module, however, was a little trickier. In fact, the latter relies, to a large extent, on the timing and the coordination of a set of distributed actions, possibly performed by Byzantine faulty processes whose behavior is hard to assess in a model checker. Instead, we use the PVS theorem prover [81] and formalize the protocol in the style of Timed Automata [4]. This formalism makes it easy to express timing constraints on transitions. It also captures several useful aspects of real-time systems such as liveness, periodicity and bounded timing delays. Using this formalism, we specified the protocol for any number of leaders, and we proved safety and liveness properties such as Proper Agreement, Agreement Termination and Integrity. Finally, the group-key management module of Enclaves is based on a secret sharing scheme whose security relies fundamentally on the hardness of computing discrete logarithms in groups of large prime order. Due to the hardness of expressing the latter correctness arguments in the logic language of a formal verification tool, we found it more convenient to give a manual proof of the module's *robustness* and *unpredictability* properties, using the Random Oracle model [9].

Finally, it is important to note that our choice for the Murphi and PVS tools, has been largely prompted by their known performance, easiness of use, and the success of a number of security-related verifications making use of them, e.g., [51, 59, 75] for Murphi and [6, 62] for PVS. The work presented in this thesis can, however, be done using other model checking and theorem proving tools.

1.4 Related Work

There has been a rich history of computer-assisted formal verification of security and fault-tolerance in distributed protocols. Some of these verifications deal with the Byzantine failure model [20], while others remain limited to the benign form [51]. Also, some of them specialized in the two-party sort of protocols (e.g. mutual authentication) deploying particularly dedicated *security logics* (e.g., BAN [16], SPC [3] and Spi calculus [1]), while others, remained more general (e.g., theorem proving and model checking) and adopted different automata formalisms to specify and reason about protocols (e.g., [7, 20, 60, 75, 89]).

The idea behind *security logics* is to formalize the epistemic reasoning of agents executing a protocol. More precisely, the logics provide constructs, axioms and inference rules for expressing simple notions of security (e.g., data secrecy) and for describing how the beliefs and knowledge of agents involved in a protocol execution evolve as messages are exchanged.

Despite their success in detecting a number of design flaws, security logics still suffer two major limitations: (1) They are often limited to special execution configurations; for instance in the original BAN one cannot express interleaved executions, yet main source of vulnerability in the Needham-Schroeder Public Key protocol [79]; (2) They often lack an appropriate semantics for epistemic-like notions such as *awareness* and *belief*.

Another notable alternative to the reasoning about security protocols is to consider protocols as sets of possible communication traces. For instance, Paulson [84] uses inductive definitions in higher-order logic to express security protocols and properties about them. In Paulson's approach [84], a protocol along with its adversarial model correspond to a set of rules expressing the observable events (i.e., message exchanges) that can be seen on the network; the property of closure under these rules yields an inductively defined set of traces modeling all possible communications between the agents as well as the intruder. Using these models, Paulson [84], interactively proves by induction that violations of security properties cannot occur in any trace. Similar logics (e.g., *awareness*-based logic) and inductive definitions are used by Accrosi *et al.* [2], Fagin *et al.* [36], and Vardi *et al.* [37]. Although inductive methods proved successful in finding attacks in security protocols, expressing security goals as inductive properties of states or traces, still remains a tricky procedure.

General-purpose automated theorem proving is the more traditional approach to verifying security properties. Early work on automated reasoning about security made use of the AFFIRM [41], Boyer-Moore [12], and Ina Jo [63] theorem-proving systems. More recently, theorem provers such as HOL [46], PVS [81], and Isabelle [83] have been used to express and reason about security protocols. These latter verification systems support specification in higher-order logic and allow users to create customized proof strategies to help automate proof search. Although major research has been put into the area of proof automation, only simple lemmas could be proved completely automatically. In other words, human guidance is still necessary for most interesting proofs.

In the same spirit as above and closer to our work, Castro and Liskov [20] specified a Byzantine fault-tolerant replication algorithm (similar to ours) using the I/O automata of Lynch and Tuttle [66]. Proofs about the algorithm's safety were manually done using invariant assertions and simulation relations. Liveness issues, however, have not been addressed. Although successful in a pen-and-paper fashion, this work has never been conducted mechanically in any theorem prover.

Kwiatkowska and Norman [60], in turn, analyzed the Asynchronous Binary Byzantine Agreement protocol [17] (based on a similar concept to our key management module) using a combination of mechanical inductive proofs (for non-probabilistic properties) and finite state checks (probabilistic properties) plus one high-level manual proof. Our approach too, takes advantage of the easiness and performance of different techniques to prove the overall Enclaves protocol.

Timed automata were also used to model the fault-tolerant protocols such as PAXOS [89] and Ensemble [50]. The authors assume a partially synchronous network and support only benign failures. This bears some similarities with verifying the Enclaves protocol in the sense that we assume some bounds on timing, but unlike the work in [50, 89], we are dealing with the more subtle Byzantine kind of failure.

In [7], Archer presented the formal verification of the TESLA (Timed Efficient Stream Loss-tolerant Authentication) broadcast authentication protocol [87] using the Timed Automata Modeling Environment (TAME). TAME provides a set of theory templates to specify and prove general I/O automata. Our work can be used, for instance, to extend the TAME package.

1.5 Outline of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we give an overview of the Enclaves protocol architecture, and describe in detail each of the modules and the services they offer. In Chapter 3, we present the model checking of the authentication module in Murphi, with views on some of the reduction techniques provided by the tool. Chapter 4 discusses the verification of the Byzantine leader agreement module in PVS. It presents how we model the elementary components of the module and how we build the final protocol model out of these ingredients. Finally, we show how we formulate and prove the correctness theorems. In Chapter 5, we briefly explain the concepts behind cryptographic reduction and sketch a mathematical proof for the robustness and unpredictability properties of the group-key management module. Chapter 6 draws major conclusions and lessons learned from this work. It also highlights some future research directions with focus on ideas to improve the automation and generalization of our work.

Chapter 2

Intrusion-Tolerant Enclaves

2.1 Introduction

Intrusion tolerance is the application of fault-tolerance methods to security. It assumes that system vulnerabilities cannot be totally eliminated, and that external attackers or malicious insiders will identify and exploit these vulnerabilities and gain illicit access to the system. The objective of intrusion tolerance is to maintain acceptable, though possibly degraded, service despite intrusions in parts of the system.

In this chapter we discuss the intrusion-tolerant version of Enclaves, a lightweight platform for building secure group applications[1]. To support such applications, Enclaves provides a secure group communication service and associated group-management and key-distribution functions. The original Enclaves system [44] and its successor [34] have a centralized architecture. An application consists of a set of group members

[1]This chapter is based in part on some of the material in [33, 34, 44]

who cooperate and communicate via a single group leader. This leader is responsible for all group-management activities, including authenticating and accepting new members, distributing cryptographic keys, and distributing group-membership information. Since the leader plays a critical role, it is an attractive target for attackers. Breaking into the leader can immediately lead to loss of confidentiality, interrupted communication, or other forms of denial of service.

The intrusion-tolerant Enclaves architecture removes this single point of failure by distributing the group management functions among n leaders. The system uses a Byzantine fault-tolerant protocol for leader coordination and a verifiable secret sharing scheme for generating and distributing cryptographic keys to group members. This new architecture is intended to tolerate the compromise of up to f leaders, where $3f + 1 < n$. Compromised leaders are assumed to be under the full control of an attacker and to have Byzantine behavior, but it is also assumed that the attacker cannot break the cryptographic algorithms used. Under these assumptions, Enclaves ensures the confidentiality and integrity of group communication, as well as proper group-management services. The remainder of this chapter describes the intrusion tolerant Enclaves in greater detail. Section 2.2 gives an overview of the architecture and design goals of Enclaves, while in Section 2.3, we present the underlying protocols and secret sharing schemes used.

2.2 Overview on Enclaves

2.2.1 Enclaves Services

A group-oriented application enables users to share information and collaborate via a communication network such as the Internet. Enclaves is a lightweight software infrastructure that provides security services for such applications [44]. Enclaves provides services for creating and managing groups of users of small to medium size, and enables the group members to communicate securely. Access to an active group is restricted to a set of users who must be pre-registered, but the group can be dynamic: authorized users can freely join, leave, and later rejoin an active application.

The communication service implements a secure multicast channel that ensures integrity and confidentiality of group communication. All messages originating from a group member are encrypted and delivered to all other members of the group. For efficiency reasons, Enclaves provides best-effort multicast and does not guarantee that messages will be received, or received in the same order, by all members. This is consistent with the goal of supporting collaboration between human users, which does not require the same reliability guarantees as distributing data between servers or computers [44].

The group-management services perform user authentication, access control, and related functions such as key generation and distribution. All group members receive a common group key that is used for encrypting group communication. A new group key is generated and distributed every time the group composition changes, that is, whenever a user enters or leaves the group. Optionally, the group key can also be refreshed on a periodic basis. Enclaves also communicates membership information to all group members. On joining the group, a member is notified of the current group composition. Once in the group, each member is notified when a new user enters or a member leaves the group. Thus, all members know who is in possession of the current group key.

In summary, Enclaves enables users to be authenticated and to join a groupware application. Once in a group, a user A is presented with a group view, that is, the list of all other group members. The system is intended to satisfy the following security requirements:

- *Proper authentication and access control*: Only authorized users can join the application and an authorized user cannot be prevented from joining the application.

- *Confidentiality of group communication*: Messages from a member U can be read only by the users who were in U's view of the group at the time the message was sent.

- *Integrity of group communication*: A group message received by U was sent by a member of U's current view, was not corrupted in transit, and is not a duplicate.

2.2.2 Centralized Architecture

The original version of Enclaves [44] and a more recent version with improved protocols [34] rely on the centralized architecture shown in Figure 2.1.

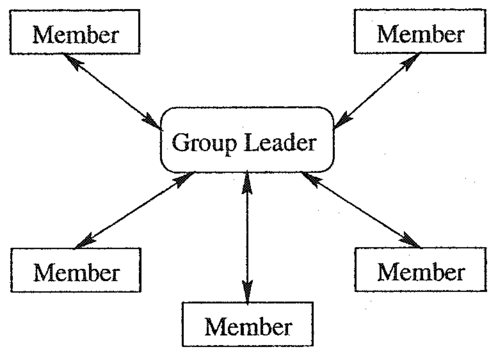

Figure 2.1: Centralized Architecture

In this architecture, a single group leader is responsible for all group-management activities. The leader is in charge of authenticating and accepting new group members, generating group keys and distributing them to members, and distributing group membership information. With such an architecture, the security requirements are satisfied if the leader and all group members are trustworthy, but the system is not intrusion tolerant. Proper service requires that the leader be trusted and never compromised. A single intrusion on the leader can easily lead to denial of service. For example, a compromised leader can interrupt group communication or prevent authorized users from joining the group. Since the leader is always in possession of the latest group key, an attacker breaking into the leader's computer can also gain access to all group communication. Increasing the resilience of the Enclaves infrastructure requires removing this single point of failure, to guarantee that group communication will remain secure even after some components of the system have been compromised.

2.2.3 Intrusion-Tolerant Architecture

The architecture of the intrusion-tolerant version of Enclaves is shown in Figure 2.2. The group and key management functions are distributed across n leaders.

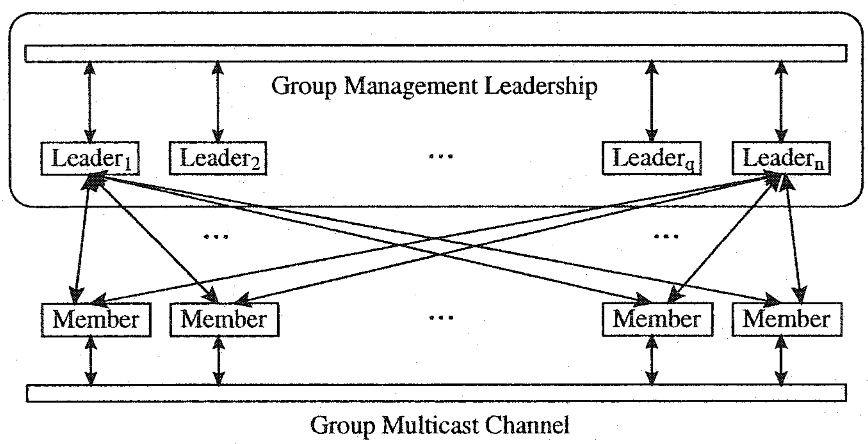

Figure 2.2: Intrusion-Tolerant Architecture

The leaders communicate with each other and with users via a synchronous network. Messages sent on this network are assumed to be eventually received, but no assumptions are made on the transmission delays and on the order of reception of messages. The architecture is designed to tolerate up to f compromised leaders, where $3f + 1 < n$. Compromised leaders are assumed to be under the full control of an attacker; They have Byzantine behavior and can collude with each other.

The security requirements are the same as previously, and we make the assumption that a fixed list of authorized participants is specified before an application starts. The new objective is now to ensure that these requirements are satisfied even if up to f leaders are compromised.

For proper group management, any modification of the group composition requires agreement between the non-faulty leaders. These leaders must agree before accepting a new member or determining that an existing member has left. Ideally, one would like all non-faulty leaders to maintain agreement on the group composition. Unfortunately, this requires solving a consensus problem, in an asynchronous network, under the Byzantine failure model. Achieving Byzantine agreement under asynchrony assumptions is a well known problem, and it has been proven that there are no deterministic algorithms to solve it [39]. Randomized algorithms (e.g., [17]) as well as algorithms relying on failure detectors (e.g., [21, 90]) are also being applied, but they tend to be complex and expensive. Instead, a weaker form of consistency property is sufficient for satisfying Enclaves's security requirements. The algorithm used in Enclaves is similar to consistent broadcast protocols such as Bracha and Toueg's protocol [13]. Combined with an appropriate authentication procedure, this algorithm ensures that any authorized user who requests to join the group will eventually be accepted. Unlike Byzantine agreement, this algorithm does not guarantee that users are accepted in the same order by all leaders. However, this does not lead to a violation of the confidentiality or integrity properties. If the group becomes stable, all non-faulty leaders eventually reach a consistent view of the group.

As in earlier implementations of Enclaves [34, 44], a common group key is shared by the group members. A new key is generated by the leaders whenever the group changes. The difficulty is to generate and distribute this key in an intrusion-tolerant fashion. All group members must obtain the same valid group key, despite the presence of faulty leaders. The attacker must not be able to obtain the group key even with the help of f faulty leaders. These two requirements are satisfied by using a secret sharing scheme proposed by Cachin *et al.* [17]. In the Enclaves framework, this scheme is used by leaders to independently generate and send individual shares of the group key to group members. The protocol is configured so that $f + 1$ shares are necessary for reconstructing the key. A share is accompanied with a description of the group to which it relates and a "proof of correctness", that is computationally hard to counterfeit. This allows group members to obtain strong evidence that a share is valid, and prevents faulty leaders from disrupting group communication by sending invalid shares.

2.2.4 Protocol Execution

Figure 2.3 shows the different phases of the protocol execution. Initially at time t_0, user U sends requests to join the group to a set of $2f + 1$ leaders. These leaders locally authenticate U within time interval $[t_1, t_2]$. When done, the agreement procedure starts and terminates at time t_4 by reaching a consensus as whether or not to accept user U. Once in the group, U remains connected to the above $2f+1$ leaders and receives key and group update messages from them. While in the group, each member is notified each time a new user joins or a member leaves the group in such a way that all members remain in possession of a consistent image of the current group-key holders.

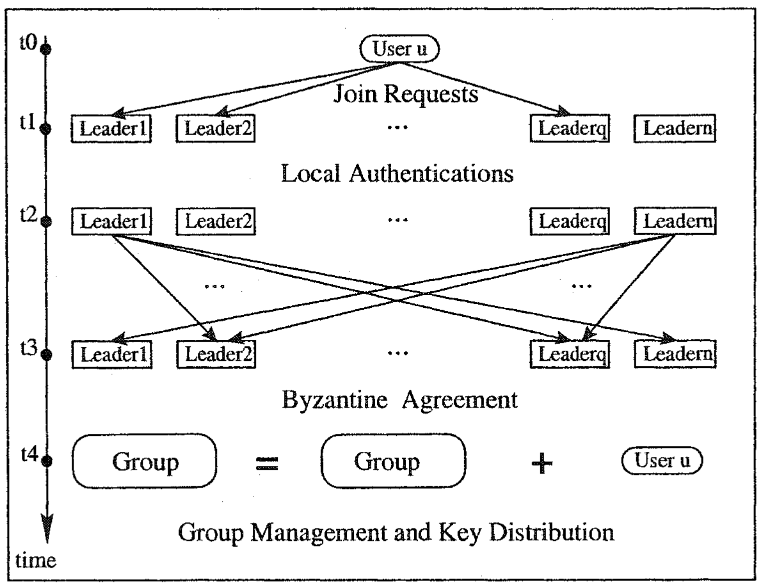

Figure 2.3: Protocol Execution

The intrusion tolerant capabilities of Enclaves rely on the combination of a cryptographic authentication protocol, a Byzantine fault-tolerant leader-coordination algorithm, and a secret sharing scheme. These protocols are presented in greater detail in next section.

2.3 Enclaves Protocols

2.3.1 Authentication

To join the group, a user U must first initiate an authentication protocol with $2f+1$ distinct leaders. U is accepted as a new group member if it is correctly authenticated by at least $f+1$ leaders. This ensures that f faulty leaders cannot prevent an honest user from joining the group, and conversely that f faulty leaders cannot allow an unauthorized user to join the group.

For authentication purposes, all users registered as authorized participants in an application share a long-term secret key with each leader. If L_i is one of the leaders, U has a long-term key $P_{U,i}$ that is known by L_i and U. The following protocol is used by U to authenticate with L_i.

$$
\begin{array}{lll}
i. & U \longrightarrow L_i : & \mathsf{AuthInitReq}, U, L_i, \{U, L_i, N_1\}_{P_{U,i}} \\
ii. & L_i \longrightarrow U : & \mathsf{AuthKeyDist}, L_i, U, \{L_i, U, N_1, N_2, K_{U,i}\}_{P_{U,i}} \\
iii. & U \longrightarrow L_i : & \mathsf{AuthAckKey}, U, L_i, \{U, L_i, N_2, N_3\}_{K_{U,i}}
\end{array}
$$

As a result of this exchange, U is in possession of a session key $K_{U,i}$ that has been generated by L_i. All group-management messages from L_i to U are encrypted with $K_{U,i}$. Thus, a secure channel is set up between U and L_i that ensures confidentiality and integrity of all group-management messages from L_i to U. Nonces and acknowledgments protect against replay attacks as discussed in [34]. The key $K_{U,i}$ is in use until U leaves the group. A fresh session key will be generated if U later rejoins the group.

2.3.2 Leader Coordination

If a non-faulty leader L_i successfully authenticates U, L_i does not immediately add U as a new group member. Instead, the leader coordination algorithm is executed (a similar algorithm is used to coordinate leaders when a member leaves the group).

Leader L_i runs the following protocol:

After successful authentication of U,

L_i sends $\langle$Propose, i, U, n_i $\rangle_{\sigma_i}$ to all leaders.

After receiving $f+1$ valid $\langle$Propose, i, U, n_j $\rangle_{\sigma_j}$

from different leaders, L_i sends $\langle$Propose, i, U, n_i $\rangle_{\sigma_i}$

to all leaders if it has not been already done so.

When L_i receives $n-f$ valid $\langle$Propose, i, U, n_j $\rangle_{\sigma_j}$ from

$n-f$ distinct leaders, L_i accepts U as a new member.

Here the notation $\langle\ \rangle_{\sigma_i}$ denotes a message digitally signed by L_i and σ_i denotes that signature. The constant n_i is used to protect against replay attacks. Each leader maintains a local integer variable n_i and its local view M_i of the current group members. M_i is updated and n_i is incremented every time L_i accepts a new member or removes an existing member. The message $\langle$Propose, i, U, n_j $\rangle_{\sigma_j}$ is considered valid by L_i if the signature checks, $n_j \geq n_i$, and U is not already a member of M_i. The pair $\langle n_i, M_i \rangle$ is L_i's current view of the group. L_i must include its own $\langle$Propose...$\rangle$ message among the $n - f$ messages necessary before accepting U.

This algorithm is a variant of existing consistent broadcast algorithms first presented in [13, 65]. It satisfies the following properties as long as no more than f leaders are faulty.

- *Consistency*: If one non-faulty leader accepts U then all non-faulty leaders eventually accept U.
- *Liveness*: If $f+1$ non-faulty leaders announce U, then U is eventually accepted by all non-faulty leaders.
- *Valid Authentication*: If one non-faulty leader accepts U then U has been announced, and thus authenticated, by at least one non-faulty leader.

The last property prevents the attacker from introducing unauthorized users into the group. Conversely, if U is an authorized user and correctly executes the authentication protocol, U will be announced by $f + 1$ non-faulty leaders, and thus will eventually be accepted as a new member by all non-faulty leaders.

The protocol works in an asynchronous network model where transmission delays are unbounded. It does not ensure that all non-faulty leaders always have a consistent group view. Two leaders L_i and L_j may have different sets M_i and M_j for the same view number $n_i = n_j$. This happens if several users join or leave the group concurrently, and their requests and the associated ⟨Propose...⟩ messages are received in different orders by L_i and L_j. If the group becomes stable, that is, no requests for join or leave are generated in a long interval, then all non-faulty leaders eventually converge to a consistent view. They communicate this view and the associated group-key shares to all their clients who also eventually have a consistent view of the group and the same group key. Temporary disagreement on the group view may cause non-faulty leaders to send valid but inconsistent group-key shares to some members. This does not compromise the security requirements of Enclaves but may delay the distribution of a new group key.

2.3.3 Group-Key Management

The group-key management protocol relies on secure secret sharing. Each of the n leaders knows only a share of the group key, and at least $f+1$ shares are required to reconstruct the key. Any set of no more than f shares is insufficient. This ensures that a compromise of at most f leaders does not reveal the group key to the attacker. In most secret sharing schemes, n shares $s_1, ..., s_n$ are computed from a secret s and distributed to n shareholders. The shares are computed by a trusted dealer who needs to know s. In Enclaves, a new secret s and new shares must be generated whenever the group changes. This must be done online and without a dealer, to avoid a single point of failure. A further difficulty is that some of the parties involved in the share renewal process may be compromised. A solution to these problems was devised by

Cachin *et al.* in [17]. In their protocol, the n shareholders can individually compute their share of a common secret s without knowing or learning s. One can compute s from any set of $f+1$ or more such shares, but f shares or fewer are not sufficient. The shares are all computed from a common value $\tilde{g}$ that all shareholders know. In our context, the shareholders are the group leaders and $\tilde{g}$ is derived from the group view using a one-way hash function. Leader L_i computes its share s_i using a share-generation function S, the value $\tilde{g}$, and a secret x_i that only L_i knows: $s_i = S(\tilde{g}, x_i)$. Leader L_i also gives a proof that s_i is a valid share for $\tilde{g}$. This proof does not reveal information about x_i but enables group members to check that s_i is valid. The protocol proposed by Cachin *et al.* [17] requires a trusted dealer to set up a number of public and private keys in an initialization phase. This can be performed off-line, and the dealer is not involved in any of the subsequent computations. The share computations are performed individually by the leaders. The share validity checks and group key construction are performed individually by group members. This protocol is related to verifiable secret sharing (e.g., [24, 38, 86]) and, more closely, to threshold signature schemes (e.g., [29, 40, 96]).

The secrecy properties of the protocol rely on the hardness of computing discrete logarithms in a group of large prime order. Such a group G can be constructed by selecting two large prime numbers p and q such that $p = 2q+1$ and defining G as the unique subgroup[2] of order q in $\mathbb{Z}_p{}^*$. The dealer chooses a generator g of G and

[2]The group $\mathbb{Z}_p{}^*$ is a cyclic group of order $p-1$, i.e., $2q$. It is a standard fact in the elementary theory of groups that : if R is a cyclic group of order m, then for every divisor d of m there exists a unique subgroup of R of order d.

performs the following operations:

- Select randomly $f+1$ elements $a_0, ..., a_f$ of $\mathbb{Z}_q$. These coefficients define a polynomial of degree f in $\mathbb{Z}_q[X]$:

$$F = a_0 + a_1 X + ... + a_f X^f.$$

- Compute $x_1, ..., x_n$ of $\mathbb{Z}_q$ and $g_1, ..., g_n$ of G as follows:

$$\begin{aligned} x_i &= F(i) \\ g_i &= g^{x_i}. \end{aligned}$$

  The numbers $x_1, ..., x_n$ must then be distributed secretly to the n leaders $L_1, ..., L_n$, respectively. The generator g and the elements $g_1, ..., g_n$ are made public. They must be known by all users and leaders.

As in Shamir's secret sharing scheme [95], any subset of $f+1$ values among $x_1, ..., x_n$ allows one to reconstruct F by interpolation, and then to compute the value $a0 = F(0)$. For example, given $x_1, ..., x_{f+1}$, one has

$$a_0 = \sum_{i=1}^{f+1} b_i x_i,$$

where b_i is obtained from $j = 1, ..., f+1$ by

$$b_i = \frac{\prod_{j \neq i} j}{\prod_{j \neq i} (j - i)}$$

By this interpolation method, one can compute $\tilde{g}^{a_0}$ for any $\tilde{g} \in G$ given any subset of $f+1$ values among $\tilde{g}^{x_1}, ..., \tilde{g}^{x_n}$. For example, from $\tilde{g}^{x_1}, ..., \tilde{g}^{x_{f+1}}$, one gets

$$\tilde{g}^{a_0} = \prod_{i=1}^{f+1} \tilde{g}_i^{\,b_i} \tag{2.1}$$

As discussed previously, leader L_i maintains a local group view $\langle n_i, M_i \rangle$. L_i's share s_i is a function of the group view, the generator g, and L_i's secret value x_i. L_i first computes $\tilde{g} \in G$ using a one-way hash function H_1:

$$\tilde{g} = H_1(n_i, M_i).$$

The share s_i is then defined as

$$s_i = \tilde{g}^{x_i}.$$

The group key for the view $\langle n_i, M_i \rangle$ is defined as

$$K = \tilde{g}^{a_0},$$

Using Equation (2.1), a group member can compute $\tilde{g}^{a_0}$ given any subset of $f+1$ or more shares for the same group view. The security of this approach has been originally proved in [17]. Under a standard intractability assumption, it is computationally infeasible to compute K knowing fewer than $f+1$ shares. It is also infeasible for an adversary to predict the values of future group keys K even if the adversary corrupts f leaders and has access to f secret values among $x_1, ..., x_n$.

Equation (2.1) allows a group member to compute the key $\tilde{g}^{a_0}$ from $f+1$ valid shares of the form $s_i = \tilde{g}^{x_i}$. However, a compromised leader L_i could make the computation fail by sending an invalid share $s_i \neq \tilde{g}^{x_i}$. L_i could also cause different members to compute different group keys by sending different shares to each. To protect against such attacks, the share s_i is accompanied with a proof of validity.

This extra information enables a member to check that s_i is equal to $\tilde{g}^{x_i}$ with very high probability. The verification uses the public value g_i that is known to be equal to g^{x_i} (since the dealer is trusted). To prove validity without revealing x_i, leader L_i generates evidence that

$$\log_{\tilde{g}} s_i = \log_g g_i.$$

This uses a technique proposed by Chaum and Pedersen [23]. To generate the evidence, L_i randomly chooses a number y in $\mathbb{Z}_q$ and computes

$$\begin{aligned} u &= g^y \\ v &= \tilde{g}^y. \end{aligned}$$

Then L_i uses a second hash function H_2 from G^6 to $\mathbb{Z}_q$ to compute

$$\begin{aligned} c &= H_2(g, g_i, u, \tilde{g}, s_i, v) \\ z &= y + x_i c. \end{aligned}$$

The proof that s_i is a valid share for $\tilde{g}$ is the tuple $\langle n_i, M_i, u, v, z)\rangle$. The information sent by L_i to a group member U is then the tuple $\langle s_i, n_i, M_i, u, v, z\rangle$. This message is sent via the secure channel established between U and L_i after authentication. This prevents an attacker in control of f leaders from obtaining extra shares by eavesdropping on communications between leaders and clients.

On receiving the above message, a group member U evaluates $g = H_1(n_i, M_i)$ and accepts the share as valid only if the following equations hold:

$$\begin{aligned} g^z &\stackrel{?}{=} u g_i^{c'} \\ \tilde{g}^z &\stackrel{?}{=} v s_i^{c'}. \end{aligned}$$

where c' (supposed to be equal to c) is given by:

$$c' = H_2(g, g_i, u, \tilde{g}, s_i, v).$$

If this check fails, s_i is not a valid share and U ignores it. Once U receives $f+1$ valid shares corresponding to the same group view, U can construct the group key. Since U maintains a connection with at least $f+1$ honest leaders, U eventually receives at least $f+1$ valid shares for the same view, once the group becomes stable.

Cachin *et al.* [17] proved that it is computationally infeasible, in the random oracle model, for a compromised leader L_i to produce an invalid share s'_i and two values c and z that pass the share-verification check.

Chapter 3

Proving Authentication by Model Checking in Murphi

Murphi [31] is a verification tool that has been successfully applied to several industrial protocols, especially in the area of multiprocessor cache coherence protocols, multiprocessor memory models and also authentication protocols [31, 51, 57, 74, 82].

To use Murphi for verification, one has to model the protocol in question in the Murphi language and then augment the model with a specification of the desired properties. Murphi has a language that supports scalable models. In a scalable model one typically starts with a small protocol configuration and gradually increases the protocol size until the verification does not terminate anymore. In many cases, errors in the general protocol (possibly with an infinite number of states) will also show up in down-scaled (finite state) versions of the protocol. The Murphi verification tool is based on explicit state enumeration and supports a number of reduction techniques such as symmetry and data independency [56, 58, 59]. The desired properties of a protocol can be specified in Murphi by means of invariants. If a state is reached where some invariant is violated, Murphi prints an error trace exhibiting the problem.

Our verification has been conducted as follows[1]. First, we formulate the protocol by identifying the protocol participants, the state variable and messages, and the key actions to be taken. Then we add an intruder to the system. In our model, the intruder is able to eavesdrop messages on the network, decrypt cipher-text when it has the appropriate keys, and generate messages using any combination of the previously gained knowledge. Finally, we state the desired correctness conditions and run the protocol for some specific size parameters.

3.1 Protocol Specification in Murphi

The Murphi language is a simple high-level language for describing and specifying protocols or, more generally, non-deterministic finite-state machines. In the following we briefly outline Murphi's main features for describing protocols and formulating properties about them.

[1]Details about the Muphi models and specification codes can be found at http://hvg.ece.concordia.ca/Research/CRYPTO/Enclaves.html

3.1.1 Modeling Protocols

Protocols in Murphi are described as sets of *states*. The *state* of a protocol model consists of the values of all global variables. In a *startstate* statement, initial values are assigned to global variables. The transition from one state to another is performed by rules. Each rule has a Boolean condition and an action, which is a program segment that is executed atomically. The action may be executed if the condition is true (i.e., the rule is enabled) and typically changes global variables, yielding a new state. Most protocol models are non-deterministic since states typically allow execution of more than one rule. For instance, in a model of a cryptographic protocol, the intruder usually has the non-deterministic choice of several messages to replay. Non-determinism is among the features that make Murphi a good candidate for reasoning about security protocols.

3.1.2 Specifying Properties

Desired properties of a protocol are specified in Murphi as state invariants, which are Boolean conditions that have to be true in every reachable state. If a state is reached in which some invariant is violated, Murphi prints an error trace consisting of a sequence of states from the start state to the state exhibiting the problem.

3.2 Local Authentication in Enclaves: System Model

The local authentication module (as shown in Figure 2.3) aims at mutual authentication between users looking to join an active or a newly created group and the group leaders. Group leaders want to be assured about the users identity in order to convince the other leaders to let them in the group, and the users in turn want to have a guarantee that they are not being fooled by some imposter (e.g., a "man in the middle" that pretends a false identity, or title, for the purpose of deception).

This module was designed to work in a malicious environment, in which messages can be overheard, replayed and created by a so-called intruder. The protocol is, however, based on the "perfect" cryptography assumption, i.e., when a message m is encrypted with some participant's public key K, then only this participant will be able to decrypt the encrypted message $\{m\}_K$.

We study the following version of the protocol:

$$
\begin{array}{lll}
i. & U \longrightarrow L_i: & \mathsf{AuthInitReq}, U, L_i, \{U, L_i, N_1\}_{P_{U,i}} \\
ii. & L_i \longrightarrow U: & \mathsf{AuthKeyDist}, L_i, U, \{L_i, U, N_1, N_2, K_{U,i}\}_{P_{U,i}} \\
iii. & U \longrightarrow L_i: & \mathsf{AuthAckKey}, U, L_i, \{U, L_i, N_2, N_3\}_{K_{U,i}}
\end{array}
$$

The user U sends a nonce N_1 (i.e., a newly generated random number) along with its identifier to Leader L_i, both encrypted with the long term key $P_{U,i}$ shared by L_i and U. Leader L_i decrypts the message and obtains knowledge of N_1. It checks U's identity in a predefined database, and then generates a nonce N_2 and a session key $K_{U,i}$ and sends the whole encrypted with the shared key $P_{U,i}$. User U decrypts the message and concludes that it is indeed talking to L_i, since only L_i was able to decrypt U's initial message containing nonce N_1 (L_i is hence authenticated). Similarly U is authenticated, in the third step of the protocol, after sending an acknowledgment including N_2 and using $K_{U,i}$.

3.2.1 Modeling Users and Leaders

First, we consider the users component, referred to as *clients* in our model. In Murphi the data structure for the *clients* is as follows:

```
const
  NumClients: 3;                -- For example
type
  ClientId: scalarset (NumClients);
  ClientStates : enum {
    C_SLEEP,                    -- Initial state
    C_WAIT,                     -- Waiting for response from leader
    C_ACK                       -- Acknowledging the session key
                      };
  Client : record
    state:  ClientStates;
    leader: AgentId;            -- Leader with whom the client starts the
  end;                          -- protocol
var
  clnt: array[ClientId] of Client;
```

The number of clients is scalable and is defined by the constant *NumClients*. The type *ClientId* is a *scalarset* of size *NumClients*, i.e., a Murphi construct used to denote a subrange like $1 \ldots NumClients$, and to enable automatic symmetry reduction on instances of that type. The state of each client is stored in the array *clnt*. In the initialization statement of the model, the local state (stored in field *state*) of each client is set to C_SLEEP, indicating that no client has started the protocol yet.

The behavior of a client is modeled with two Murphi rules. The first rule is used to start the protocol by sending the initial message to some agent (supposedly a leader), and then changes its local state from C_SLEEP to C_WAIT. The second rule models the reception and checking of the reply from an agent, the commitment and the sending of the final message. The Murphi model for the first rule is as follows:

```
ruleset i: ClientId do
  ruleset j: AgentId do
    rule "client starts protocol (step 1)"
      clnt[i].state = C_SLEEP &
      !ismember(j,ClientId) &          -- only leaders and intruders
      multisetcount (l:net, true) < NetworkSize
    ==>
    var
      outM: Message;                   -- outgoing message
    begin
      undefine outM;
      outM.psource  := i;
      outM.pdest    := j;
      outM.mType    := M_AuthInitReq;
      ...
      multisetadd (outM,net);
      clnt[i].state  := C_WAIT;
      clnt[i].leader := j;
    end;
  end;
end;
```

The condition of the rule is that client i is in the local state C_SLEEP, that agent j is not trivially a client (and hence should be either a leader or an intruder), and that there is space in the network for an additional message. The network is modeled by the shared variable *net*. Once the rule is enabled, the outgoing message is constructed and added to the network. In addition, the local state is updated and the identifier of the intended destination is stored in state variable *clnt*[*i*].*leader*.

The second rule of the client is modelled in Murphi as follows:

```
ruleset i: ClientId do
  choose j: net do
    rule "client reacts to nonce received (steps 2/3)"
      clnt[i].state = C_WAIT &
      net[j].pdest = i &
      ismember(net[j].psource,IntruderId)
    ==>
    var
      outM: Message;   -- outgoing message
      inM:  Message;   -- incoming message
    begin
      inM := net[j];
      multisetremove (j,net);

      if inM.key=i then             -- message is encrypted with i's key
        if inM.mType= M_AuthKeyDist then    -- correct message type
          if (inM.nonce1=i &                -- correct nonce and source
             inM.csource=clnt[i].leader) then
            undefine outM;
            outM.psource  := i;
            outM.csource  := i;
            outM.pdest    := clnt[i].leader;
            outM.cdest    := clnt[i].leader;
            outM.key     := inM.sessionKey;
            outM.mType   := M_AuthAckKey;
            outM.nonce1  := inM.nonce2;

            multisetadd (outM,net);
            clnt[i].state := C_ACK;
          else
            --error "Client received incorrect nonce"
          end;
    ......
end;
```

The condition of the rule is that client i is in the local state C_WAIT and that the destination of the message in network cell $net[j]$ is actually this client. Once enabled, the rule recovers the message and frees the network. The client then checks if it can decrypt the message, if the message type is correct, and if the message contains the correct nonce. If all three conditions hold, an outgoing acknowledgment message $M\_AuthAckKey$ is constructed and added to the network. The client then changes its local state to C_ACK.

In this model, we do not consider possible interferences that may occur between sessions (e.g., when a client U concurrently sends several authentication requests to different leaders). Instead, we assimilate the protocol to a sequence of independently running parallel sessions, and focus, rather, on the mutual authentication between each single leader L_i and client U.

The leader part of the model is quite similar to the client part. For instance, the leaders also maintain a local state and store the identifier of the agent initiating the protocol in their state variable $lead[i].client$. In addition, the behavior of the leaders is also modeled with two rules: one that handles the initial authentication request of the client and another which commits to the session after receipt of the final message of the protocol.

3.2.2 Modeling Intruders

The intruder maintains a set of overheard messages and an array representing all the nonces it knows. The behavior of the intruder is modeled with three rules: one for eavesdropping and intercepting messages, one for replaying messages, and one for generating messages using the learned nonces and injecting them into the network. The model for the first rule is given in the following.

```
ruleset i: IntruderId do
  choose j: net do
      rule "intruder overhearing messages"
        !ismember (net[j].psource, IntruderId) -- not for intruder' msgs
      ==>
      var
        temp: Message;

      begin
        alias msg: net[j] do    -- message to intercept
        alias intruderknowledge: int[i].messages do
        if multisetcount (f:intruderknowledge, true) < MaxKnowledge then
          if msg.key=i then    -- message is encrypted with i's key
            int[i].nonces[msg.nonce1] := true;      -- learn nonces
            if msg.mType= M_AuthKeyDist  then
              int[i].nonces[msg.nonce2] := true;
            end;
          else                                      -- learn message
            alias messages: int[i].messages do
              temp := msg;
              undefine temp.psource;    -- delete useless information
              undefine temp.pdest;
              if multisetcount (l:messages,    -- add only if not there
                   messages[l].mType = temp.mType &
                    .....
                   (messages[l].mType = M_AuthKeyDist ->
                   messages[l].sessionKey = temp.sessionKey ) ) = 0 then
                multisetadd (temp, int[i].messages);
              end;
      ........
end;
```

The enabling condition of the *intruder' message overhearing* rule is that the network cell in question, $net[j]$, does not contain a message sent by the intruder itself (otherwise nothing will be learned). We distinguish then two cases:

- The intercepted message is intended for the intruder (encrypted with a key known to the intruder $msg.key = i$), then the action is simply to learn the nonces (c.f. murphi model above).

- The intruder intercepts a message that is intended to another participating agent and then learns all useful message fields. The intruder can also be modeled to block and remove messages from the network.

3.2.3 Properties Specification

The main property we are interested in is *mutual authentication* between a given pair of leader and client, L_i should be able to assert that it has been talking, indeed, to client U, and vice-versa. The verification is done by means of invariant checking under the aforementioned assumptions. The *client proper authentication* invariant is given below.

```
invariant "client proper authentication"
  forall i: LeaderId do
    lead[i].state = L_COMMIT &
    ismember(lead[i].client, ClientId)
    ->
    clnt[lead[i].client].leader = i &
    clnt[lead[i].client].state = C_ACK
  end;
```

It basically states that for each leader i, if it committed to a session with a client, then this client (whose identifier is stored in *lead*[i].*client*), must have started the protocol with leader i, i.e., have stored i in its field *leader* and be awaiting for acknowledgment (i.e., in state C_ACK).

In addition to the above invariant, we have checked a similar one for *leaders proper authentication.* The *leaders proper authentication* invariant asserts that for each client, if it commits to a session with a leader L_i, then L_i is, in reality, the same leader with whom the client started the session.

```
invariant "leaders proper authentication"
 forall i: ClientId do
   clnt[i].state = C_ACK &
   ismember(clnt[i].leader, LeaderId)
   ->
   lead[clnt[i].leader].client = i &
   ( lead[clnt[i].leader].state = L_WAIT |
     lead[clnt[i].leader].state = L_COMMIT )
 end;
```

3.3 Experimental Results

In the previous sections, we have discussed the formalization in Murphi of Enclaves *local authentication* module. We have presented our model assumptions, defined the rules that will be used by the protocol agents, and specified two security properties as invariants of the protocol. In the following, we present the experimental results obtained from the model checking of the first invariant: *clients proper authentication.*

Table 3.1: Model Checking Experimental Results

# Clients	# Leaders	# Intruders	Network size	States	CPU time
2	2	1	1	274753	515 s
3	2	1	1	–	–
2	3	1	1	1240550	3408 s
2	4	1	1	3723157	18383 s
2	5	1	1	–	–
3	2	1	1	–	–
3	1	1	1	1858746	3161 s
2	2	2	1	–	–
2	2	1	2	–	–
3	1	1	2	–	–
3	1	2	1	–	–
4	6	1	1	–	–
⋮	⋮	⋮	⋮	⋮	⋮
4	2	3	6	–	–

Table 3.1 shows the number of reached states and CPU run times taken on a six-440-MHz-processor Sun Enterprise Server with 6 GB of memory, for different sizes of the protocol. The instances of the protocol that we have considered, were chosen in a way that emphasizes the weight of each size parameter. Our approach is as follows. We start with an instance of the protocol for which the model checking terminates (e.g., the first row in the table), and from there we explore several instances, following a certain pattern, where we vary only one size parameter and keep all others unchanged. The results roughly show that the number of leaders is less significant, in terms of complexity, then other parameters such as the number of

clients, intruders or the network size (maximum number of messages allowed on the network at the same time). This can be explained by the fact that the average load for each individual leader is reduced when we increase their total number. Another parameter, of most importance, is the intruder's maximum knowledge (or memory size). For the purpose of this experiment, we have kept it equal to "3" messages (enough for the three steps required in a single session of the protocol).

Besides, many of the rows in Table 3.1, show non conclusive results, where Murphi ends up running out of memory before reaching all possible states. This is a very known problem of model checking in general. One way to improve this, is by deploying more computational resources, but doing so will not bring a major change to the picture, as the number of states grows exponentially with respect to the size parameters. Another successful way is to adopt powerful model abstraction and reduction techniques [27], which in our case would have to be done manually as Murphi does not support such algorithms.

3.4 Concluding Remarks

In this chapter, we have verified the correctness of Enclaves authentication protocol using the Murphi model checker. Our model, assumes a powerful intruder, capable of intercepting messages, modifying and generating new messages out of the previously gained knowledge. We also assume perfect cryptography and, therefore, limit the intruder power by cryptographic constraints.

In this verification, we are uniquely concerned with the *mutual authentication* property between any given pair of client and leader. In the case of a client concurrently running several authentication requests with different leaders, we consider each session (with a given leader) as independent. In other words, we assume that the correctness of the concurrently running sessions results from the correctness of each session running independently.

Besides, our specification is very intuitive; we simply list the rules of each action the participants can perform in the protocol. Unlike belief-based logics, we do not need to interpret the beliefs that each message would convey to protocol agents. Also Murphi offers a lot of freedom, compared to other tools (e.g., SMV [60]), especially when it comes to the definition of the intruder model, often at the center of all security analysis.

On the other hand, efficiency was our biggest concern. In fact, only for a few number of small instances of the protocol, the Murphi tool terminated the model checking in a reasonable amount of time, but it was not the case for most instances. Although we performed our experiments on a relatively powerful machine, and despite the fact that Murphi was equipped with a few techniques to help reduce the state space, the execution time increased dramatically as we started increasing the protocol size, and the model checker was unable to terminate any more. This is due to the exponential complexity of the protocol in the number of participants, the network size and the maximum knowledge participants are allowed to remember. One possible way to improve this, is by using rank functions with theorem proving [94].

Chapter 4

Proving Byzantine Agreement by Theorem Proving in PVS

PVS [81] is a formal specification and verification system. It is mainly intended for the formalization and verification of design specifications in hiher-order logic, and for the analysis of difficult problems. It has been chiefly applied to algorithms and architectures for fault-tolerant flight control systems, and to problems in hardware and real-time system design (e.g., [6, 62]).

PVS consists of a specification language, a number of predefined theories, and a theorem prover. The specification language of PVS is based on a classical, typed higher-order logic. The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. To use PVS, one needs first to model the considered system in the specification language of PVS, and then apply the right axioms and inference procedures to derive the proofs.

In our verification of Enclaves, we model the protocol as an automaton whose initial state is modified by the participants' actions as the group mutates (new members join). Moreover, because Enclaves depends also on time (participants timeout, timestamp group views, etc.), we found it convenient to model it as a timed automaton. In the current specification, timing is used only to ensure actions progress. Timing, however, is essential to prove upper bounds on agreement delays (e.g., a maximum join delay), but this is beyond the scope of this thesis. Participants in a typical run of Enclaves consist of a set of n leaders (f of which are faulty), a group of members, and one or more users requiring to join the group.

In this chapter[1], we first explain our general PVS theory about timed automata. The parameters of this theory are used here to formalize Enclaves by defining the actions, the states, and the precondition and effect of each action. Thereafter, we describe the resulting executions of the protocol and fault assumptions.

[1]Details about the PVS theories and proof scripts can be found at http://hvg.ece.concordia.ca/Research/CRYPTO/Enclaves.html

4.1 Modeling Byzantine Agreement in PVS

4.1.1 Timed Automata

We have developed a general, protocol-independent, PVS theory called *TimedAutomata*. Given a number of parameters, it defines all possible executions of the protocol as a set of *Runs*. A *run* is a sequence of the form $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots$ where the s_i are *states*, representing a snapshot of the system during execution and the a_i are the executed *actions*. A particular protocol (an instance of the timed automaton) is characterized by sets of possible *States* and *Actions*, a condition *Init* on the initial state, the precondition *Pre* of each action, expressing in which states that action can be executed, the effect *Effect* of each action, expressing the possible state changes by the action, and a function *now* which gives the current time in each state.

In a typical application, there is a special *delay* action which models the passage of time and increases the value of *now*. All other actions do not change time. In PVS, the theory and its parameters are defined as follows.

```
TimedAutomata  [ States, Actions: TYPE+,
                 Init : pred[States],
                 Pre : [Actions -> pred[States]] ,
                 Effect  :  pred[[States, Actions, States]],
                 now : [States -> nonneg_real]
               ] : THEORY
```

To define *Runs*, let *PreRuns* be a record with two fields, *states* and *events*.

```
PreRuns : TYPE = [# states   : sequence[States],
                    events   : sequence[Actions] #]
```

A *Run* is a *PreRun* where the first state satisfies *Init*, the precondition and effect predicates of all actions are satisfied, the current time never decreases and increases above any arbitrary bound (avoiding Zeno-behaviour [49]). In PVS, this is formalized as follows.

```
PreEffectOK(pr) : bool = FORALL i :
   Pre(events(pr)(i)) (states(pr)(i)) AND
   Effect(states(pr)(i), events(pr)(i), states(pr)(i + 1))


NoTimeDecrease(pr) : bool =
   FORALL i : now(states(pr)(i)) <= now(states(pr)(i + 1))

NonZeno(pr): bool =
   FORALL t :   EXISTS i :  t < now(states(pr)(i))

Runs : TYPE =
   { pr: PreRuns | Init(states(pr)(0)) AND PreEffectOK(pr) AND
                   NoTimeDecrease(pr) AND NonZeno(pr) }
```

4.1.2 Modeling Leaders Actions

To define the actions of the leaders, we first state a few preliminary definitions. Let n be the number of leaders and let f be such that $3f+1 \leq n$ (the maximum number of faulty leaders).

```
   n  : posnat
   f : { k : nat | 3 * k + 1 <= n }
```

For simplicity, leaders are identified by an element of $\{0, 1, \ldots, n-1\}$. Users are represented by some uninterpreted non-empty type, and time is modeled by the set of non-negative real numbers. Besides, we define three time constants for the maximum delay of messages in the network, and the maximum delays between some leader actions.

```
LeaderIds : TYPE = below[n]
UserIds   : TYPE+
Time :      TYPE+ = nonneg_real

i       : VAR LeaderIds
user    : VAR UserIds
t       : VAR Time
MaxMessageDelay, MaxTryPropagate, MaxTryAccept  : Time
```

We define the following actions of the protocol:

- A general *delay* action, which occurs in all our timed models; it increases the current time (*now*), and all other clocks that may be defined in the system, with the amount specified by a delay parameter *del*.
- An *announce* action used to send announcement messages of new locally authenticated users to the other leaders of the protocol.
- A *trypropagate* action allowing a user announcement to be further spread among leaders. This action is executed periodically, but it only changes the state of the system if enough announcements ($f + 1$) have been received for the considered user and it has not already been announced or propagated by the leader in question before.
- An action *Tryaccept* used to let leaders periodically check whether they have received enough announcements and/or propagation messages for a given user. Once this condition is satisfied, the user is accepted to join the group.

- A *receive* action allowing a leader to receive messages; it removes a received message from the network and adds corresponding data to the local buffer of the leader.

- A *crash* action modeling the failure of a leader. After a crash, a leader may still perform all the actions mentioned above, but in addition it may perform a *misbehave* action.

- An action *misbehave*, which models the Byzantine mode of failure and can only be performed by a faulty (crashed) leader.

These actions are represented in PVS as a data type, which ensures, e.g., that all actions are syntactically different.

```
LeaderActions [LeaderIds, UserIds, Time : TYPE] : DATATYPE
BEGIN
   delay(del : Time) : delay?
   announce(id : LeaderIds, user : UserIds) : announce?
   trypropagate(id : LeaderIds) : trypropagate?
   tryaccept(id : LeaderIds) : tryaccept?
   receive(id : LeaderIds) : receive?
   crash(id : LeaderIds) : crash?
   misbehave(id : LeaderIds) : misbehave?
END LeaderActions
```

4.1.3 Modeling States

In order to properly capture the distributed nature of the network, it is suitable to model two kinds of states: a *local* state for each leader, accessible only to the particular leader, and a *global* state to represent global system behavior which includes the local state of each leader, the representation of the network and a global notion of time.

An important part of the local state is the group *view*, which is a set of users in the current group. In fact, the ultimate goal of Enclaves is to assure consistency of the group views. Moreover, we use a Boolean flag (*faulty*) marking the leader status as faulty or not, some local timers (*clockp* and *clocka*) to enforce upper bounds on the occurrence of *trypropagate* and *tryaccept* actions, and finally a list (*received*) of the leaders from which the local leader received proposals for a given user.

```
Views : TYPE = setof[UserIds]

LeaderStates : TYPE =
  [# view          : Views,
     faulty        : bool,
     clockp        : Time,    % clock for the trypropagate action
     clocka        : Time,    % clock for the tryaccept action
     received      : [UserIds -> list[LeaderIds]]   #]
```

We model *Messages* as quadruples containing a source, a destination, a proposed user and a timestamp indicating an upper bound on the delivery time, i.e., the message must be received before the *tmout* value.

```
Messages : TYPE = [# src      : LeaderIds,
                     tmout    : Time,
                     proposal : UserIds,
                     dest     : LeaderIds   #]
```

In the *global states*, the network is modeled as a set of messages. Messages that are broadcast by leaders are added to this set, with a particular time-out value, and they are eventually received, possibly with different delays and at a different order at recipient ends. The global state also contains the local state of each leader and a global notion of time, represented by *now*.

```
GlobalStates : TYPE = [# ls      : [LeaderIds -> LeaderStates],
                         now     : Time,
                         network : setof[Messages]  #]
s, s0, s1    : VAR GlobalStates
```

Furthermore, we define a predicate *Init*, that expresses conditions on the initial state, requiring that all views, received sets and the network are empty, and all clocks and *now* are set to zero.

4.1.4 Preconditions and Effects

For each action A, we define its precondition, expressing when the action is enabled, and its effect.

```
Pre(A)(s) : bool =
   CASES A OF
     delay(t)        : prenetwork(s,t) AND preclock(s,t),
     announce(i,u)   : true,
     trypropagate(i) : true,
     tryaccept(i)    : true,
     receive(i)      : MessageExists(s,i),
     crash(i)        : NOT faulty(ls(s)(i)),
     misbehave(i)    : faulty(ls(s)(i))
   ENDCASES
```

An *announce* action may always occur and hence has precondition *true*. Similarly for *trypropagate* and *tryaccept*, which should occur periodically. Action *receive*(i) is only allowed when there exists a message in the network with destination i. For simplicity, a *crash* action is only allowed if the leader is not faulty (but we could also take a precondition *true*). A *misbehave* action may only occur for faulty leaders.

Most interesting is the precondition of the *delay*(t) action. This action increases *now* and all timers (*clockp* and *clocka*) by t. To ensure that messages are delivered before their time-out value, we require that the condition *prenetwork*, defined below, holds in the state before any *delay*(t) action is taken, which fits our informal assumptions about network reliability.

```
prenetwork(s, t) : bool =  FORALL msg :
    member(msg, network(s)) IMPLIES  now(s) + t <= tmout(msg)
```

Similarly, there is a condition *preclock* which requires that all timers (*clockp* and *clocka*) are not larger than *MaxTryPropagate* and *MaxTryAccept*, respectively. Since the *trypropagate* and *tryaccept* actions reset their local timers to zero, this may enforce the occurrence of such an action before a time delay is possible.

Next, we define the effect of each action, relating a state s_0 immediately before the action and a state s_1 immediately afterwards.

- *delay*(t) increments *now* and all local timers by t, as defined by $s_0 + t$.
- *announce*(i, U) adds, for each leader j a message to the network, with source i, time-out *now*(s_0) + *MaxMessageDelay*, proposal U, and destination j.
- *trypropagate*(i) resets *clockp* to zero and adds to the network messages, to all leaders, containing proposals for each user for which at least $f + 1$ messages have been received.

- *tryaccept*(i) resets *clocka* to zero and adds to its local view all users for which at least $(n - f)$ messages have been received.
- *receive*(i) removes a message with destination i from the network, say with source j and proposal U, and adds j to the list of received leaders for U, provided it is not in this list already.
- *crash*(i) sets the flag *faulty* of i to *true*.
- *misbehave*(i) may just reset the local timers *clockp* and *clocka* of i to zero, as expressed by $ResetClock(s_0, i, s_1)$ below, or it may add randomly as well as maliciously chosen messages to the network (provided that timeouts are not violated). A misbehaving leader, however, cannot impersonate other protocol participants, i.e., any message sent on the network has the identifier of its actual sender.

This leads to a predicate of the form:

```
Effect(s0,A,s1) : bool =
   CASES A OF
     delay(t)         : s1 = s0 + t,
     announce(i,u)    : AnnounceEffect(s0,i,u,s1),
     ...
     misbehave(i)     : ResetClock(s0,i,s1) OR SendMessage(s0,i,s1)
   ENDCASES
```

4.1.5 Protocol Runs and Fault Assumptions

Runs of this timed automata model of Enclaves are obtained by importing the general timed automata theory. This leads to type *Runs*, with typical variable r. Let

$Faulty(r, i)$ be a predicate expressing that leader i has a state in which it is faulty. It is easy to check in PVS that once a leader becomes faulty, it remains faulty forever. Let $FaultyNumber(r)$ be the number of faulty leaders in run r (it can be defined recursively in PVS). Then we postulate by an axiom that the maximum number of faults is f ($MaxFaults\ :\ AXIOM\ FaultyNumber(r) \leq f$).

4.2 Correctness Proofs

One of the most important requirements Enclaves is supposed to guarantee, is *proper authentication and access control.* This requirement ensures that only authorized users can join the group and that an authorized user cannot be prevented from joining the group. In this chapter, we are interested in verifying this requirement, formalized hereafter by the three following properties:

- **Termination**: if user U wants to join an active group and has been announced by enough non-faulty leaders, then eventually user U will be accepted by all non-faulty leaders and become a member of the group.

- **Integrity**: a user that has been accepted in the group should have been announced by a non-faulty leader earlier during the protocol execution.

- **Proper Agreement**: if a non-faulty leader decides to accept user U, then all non-faulty leaders accept user U too.

The above properties do only reflect the author's perception and understanding of the *proper authentication and access control* requirement. To the best of our knowledge, there is no method available in the literature that allows to rigorously and "losslessly" formalize an informal assertion. In the remainder of this section, we assume the above properties to convey the full meaning of the requirement in question, and we briefly outline each of the properties and their respective proofs.

Theorem 1 (Termination)

For all r and U, $announced\_by\_many(r, U)$ implies $accepted\_by\_all(r, U)$

where

- $announced\_by\_many(r, U)$ expresses that at least $(f + 1)$ non-faulty leaders announced user U during run r;
- $accepted\_by\_all(r, U)$ asserts that eventually all non-faulty leaders have user U in their *view* during run r.

Proof

Assume $announced\_by\_many(r, U)$, which implies that at least $(f + 1)$ non-faulty leaders broadcast a proposal for U. Because of the reliability of the network, eventually these messages will be delivered to their destination, and in particular to the $(n - f)$ non-faulty leaders of the network. They all receive $(f + 1)$ announcement messages for user U, which is enough to trigger the propagation procedure (for U) for all non-faulty leaders who did not participate in the announcement phase. Now because of the network reliability, we conclude that eventually all non-faulty leaders will receive at least $(n - f)$ approvals for user U, enough to make a majority, since $(n - f) > f$ follows from $n > 3f$. □

Theorem 2 (Integrity)

For all r and U, $accepted\_by\_one(r, U)$ implies $announced\_by\_one(r, U)$

where

- $accepted\_by\_one(r, U)$ holds if at least one leader eventually included U in its *view* during run r.
- $announced\_by\_one(r, U)$ expresses that at least one non-faulty leader announced user U during run r;

Proof

We proceed by contrapositive and use the non-impersonation property. We assume that for all non-faulty leaders no announcement for user U has been done during run r. Now because of non-impersonation, faulty leaders cannot send more than f different announcements. This implies that the leaders would receive no more than f announcements for user U, which is not enough to trigger propagation actions. This yields that U will never be proposed by any of the non-faulty leaders, and hence none of them will receive as much as $(n - f)$ messages for U (recall $(n - f) > f$). As a result, user U will never be accepted by any of the non-faulty leaders. □

Theorem 3 (Proper Agreement)

For all r and U, $accepted\_by\_one(r, U)$ implies $accepted\_by\_all(r, U)$

Proof

$accepted\_by\_one(r, U)$ implies that there exists a non-faulty leader that received at least $(n - f)$ approvals (i.e., announcements or propagation messages) for user U. Among these approvals, at least $(n - 2f)$ come from non-faulty leaders (by non-impersonation). Now because these leaders are non-faulty, they broadcast the same approval to all the other leaders. In addition, because of the network reliability, these messages are eventually delivered to destination. This implies that all $(n - f)$ non-faulty leaders receive eventually the above $(n-2f)$ approvals. Since $(n-2f) \geq (f+1)$, all $(n-f)$ non-faulty leaders have received at least $(f+1)$ messages for U. Similar to the proof of Termination, the latter implies the start of the propagation procedure, then the reception of at least $(n-f)$ approvals for user U, and finally the acceptance of U by all non-faulty leaders. $\square$

4.3 Concluding Remarks

In this chapter, we have verified the correctness of the Byzantine Agreement module of Enclaves using the the PVS theorem prover. Thanks to the high level of expressiveness of the Timed-Automata formalism, as well as the rich datatype package of PVS, we have succeeded to formalize the module for any number of leaders, in a way that thoroughly captures the many subtleties on which the correctness arguments of Enclaves rely.

Also, the PVS theorem prover provides a collection of powerful primitive inference procedures to help derive theorems. These primitive procedures can also be combined to yield higher-level proof strategies, while proofs may produce scripts that can be edited, attached to additional formulas, and rerun. This allows many similar theorems to be proved efficiently, permits proofs to be adjusted economically to follow changes in requirements or design, and encourages the development of readable proofs.

Using all these features, we have proved the protocol to satisfy its requirements of *Termination*, *Integrity* and *Proper Agreement*. The proofs required over 60 intermediate lemmas. The *Integrity* and *Termination* theorems were the most challenging to prove and they helped, by reusing some the proof scripts, deduce *Proper Agreement*.

Chapter 5

Proving Group-Key Management by Mathematical Analysis

In the previous chapters we discussed authentication and leaders agreement. We saw also that once the leaders agree on accepting a client U, they proceed with providing it with a group key. We direct our focus here to Enclaves group key management module [33]. This module, based on a secret sharing scheme, ensures that (1) the f dishonest leaders cannot obtain the group key even if they conspire altogether (at least $(f+1)$ shares are needed to reconstruct the secret); (2) the group key is renewed every time the group changes (new join or leave); and (3) the clients are able to discern valid key shares from fake ones (possibly issued by malicious leaders).

The group key management protocol of Enclaves is based on a previous work of Cachin *et al.* [17], where the authors make use of threshold signatures [96] and coin-tossing [14] schemes in order to achieve Byzantine agreement in a completely asynchronous network.

The security properties of the group key management protocol of Enclaves rely on the hardness of computing discrete logarithms in a group of large prime order. Such a group G can be constructed by selecting two large prime numbers p and q such that $p = 2q + 1$ and defining G as the unique subgroup of order q in $\mathbb{Z}_p{}^*$.

The remainder of this chapter is organized as follows. We start by a brief outline of the secret sharing scheme used in the Group-Key Management module of Enclaves. Then, after a quick survey of the state-of-the-art main models used in security analysis, we discuss our adopted model and argue for our choice. Finally, we prove the *Confidentiality of group communication* requirement via a mathematical analysis of the *robustness* and *unpredictability* properties of the group key management module of Enclaves.

5.1 Enclaves Group-Key Management: A Verifiable Secret Sharing Scheme

As discussed in Chapter 2, in order to preserve the integrity and the confidentiality of communications, the group encryption key is split into n shares $s_1, ..., s_n$. These shares are computed from a secret s and distributed to the n leaders (share holders). The shares are computed, at the initialization phase, by a trusted dealer who needs to know s. Each of the n leaders knows only a share of the group key, and at least $(f+1)$ shares are required to reconstruct the key. Any set of no more than f shares is insufficient. This ensures that compromise of at most f leaders does not reveal the group key to the attacker. The shares are all computed from a common value $\tilde{g}$ that all share holders know. The value of $\tilde{g}$ is derived from the group view using a one-way hash function. Leader L_i computes its share s_i using a share-generation function S, the value $\tilde{g}$, and a secret x_i that only L_i knows: $s_i = S(\tilde{g}, x_i)$. Leader L_i also gives a proof that s_i is a valid share for $\tilde{g}$. This proof does not reveal information about x_i but enables group members to check that s_i is valid.

Initially, the dealer chooses a generator g of G and performs the following operations:

- Select randomly $(f+1)$ elements $a_0, ..., a_f$ of $\mathbb{Z}_q$. These coefficients define a polynomial of degree f in $\mathbb{Z}_q[X]$:

$$F = a_0 + a_1 X + ... + a_f X^f.$$

- Compute $x_1, ..., x_n$ of $\mathbb{Z}_q$ and $g_1, ..., g_n$ of G as follows:

$$\begin{aligned} x_i &= F(i) \\ g_i &= g^{x_i}. \end{aligned}$$

  The numbers $x_1, ..., x_n$ must then be distributed secretly to the n leaders $L_1, ..., L_n$, respectively. The generator g and the elements $g_1, ..., g_n$ are made public. They must be known by all users and leaders.

In addition, leader L_i maintains a local group view $\langle n_i, M_i\rangle$. L_i's share s_i is a function of the group view, the generator g, and L_i's secret value x_i. L_i first computes $\tilde{g} \in G$ using a one-way hash function H_1 with domain in G:

$$\tilde{g} = H_1(n_i, M_i).$$

Then the share s_i is obtained by exponentiation:

$$s_i = \tilde{g}^{x_i}.$$

Finally, the group key for the view $\langle n_i, M_i\rangle$ is defined as

$$K = \tilde{g}^{a_0},$$

Now given any subset of $(f+1)$ values among $\tilde{g}^{x_1}, ..., \tilde{g}^{x_n}$, one can compute

$$\tilde{g}^{a_0} = \prod_{i=1}^{f+1} \tilde{g_i}^{b_i} \tag{5.1}$$

where b_i is obtained from $j = 1, ..., (f+1)$ by

$$b_i = \frac{\prod_{j \neq i} j}{\prod_{j \neq i}(j - i)}$$

It can be seen from Equation (5.1), that a compromised leader L_i can make the computation fail just by sending an invalid share $s_i \neq \tilde{g}^{x_i}$ to some client. L_i could also cause different members to compute different group keys by sending different shares to each participant. In order to protect against such attacks, the share s_i should be accompanied by a proof of validity. This extra information enables a member to check that s_i is equal to $\tilde{g}^{x_i}$ with very high probability. The verification uses the public value g_i that is known to be equal to g^{x_i} (since the dealer is trusted). To prove validity without revealing x_i, leader L_i generates evidence that

$$\log_{\tilde{g}} s_i = \log_g g_i.$$

To produce the evidence:

1. Leader L_i picks up randomly $y \in \mathbb{Z}_q$ and computes

$$\begin{aligned} u &= g^y \\ v &= \tilde{g}^y. \end{aligned}$$

2. Leader L_i, then uses a second public hash function $H_2 : G^6 \to \mathbb{Z}_q$ to compute $c = H_2(g, g_i, u, \tilde{g}, s_i, v)$.

3. Now leader L_i computes $z = (y + x_i c)$ and sends each client the tuple (n_i, M_i, s_i, u, v, z), that is the share s_i and the proof of validity (n_i, M_i, u, v, z).

4. On receiving the above message, a group member U evaluates $g = H_1(n_i, M_i)$ and computes $c' = H_2(g, g_i, u, \tilde{g}, s_i, v)$ (supposed to be equal to c) and accepts the share s_i only if the following equations hold:

$$g^z \stackrel{?}{=} u\ {g_i}^{c'} \tag{5.2}$$

$$\tilde{g}^z \stackrel{?}{=} v\ {s_i}^{c'} \tag{5.3}$$

If this check fails, s_i is not a valid share and U ignores it. Once U receives $(f + 1)$ valid shares corresponding to the same group view, U can construct the group key. Since U maintains a connection with at least $(f+1)$ honest leaders, U eventually receives at least $(f+1)$ valid shares for the same view, once the group becomes stable.

5.2 Provable Security : Approaches and Models

5.2.1 Concrete vs. Asymptotic Approaches

Approaches to provable security are mainly of two flavors: *concrete* (or exact) security and *asymptotic* security. The latter adopts a more general way in considering security: objects of interest are always families of functions. Asymptotic security is well-suited for understanding the basic theoretical relationships between cryptographic primitives. For example, asymptotic security has been used to reason about results like "if a one-way function exists, then we can build a pseudo-random generator" via some complex reductions. So similar assertions about the existence of reductions are well-captured in the asymptotic approach.

In contrast, the exact approach does not involve reasoning about families of functions, and instead, it directly models the underlying primitives of cryptographic protocols. This offers a finer assessment of security and allows to know exactly how much security is maintained by the reduction. In practice, to quantify the reduction one would define the advantage $Adv(I)$ that computationally bounded adversary I will defeat some security goal of the scheme in question. The advantage is then

$$Adv(I) = \Pr\,[I \text{ defeating the scheme}] - \tfrac{1}{2}$$

where $\frac{1}{2}$ is the probability, in a uniform model, of I defeating the scheme.

5.2.2 Random Oracle vs. Standard Models

In general, to show a cryptosystem is secure, cryptographers choose a model for analyzing the security of the cryptosystem. Models used by cryptographers fall into three main classes: *ad hoc*, *random oracle*, and *standard* models.

Ad hoc Models

As their name suggests, these models are very much of a hack. One could build an *ad hoc* model only by assembling some hash functions along with some random generators, etc. and then see if it captures few obvious attacks. If so, the system is deployed. Once broken, some hash functions along with some random generators are added and the cycle repeats.

Obviously, this approach has serious limitations (in fact, it leads nowhere). Even if the cryptosystem is built out of perfect components, these components may interact in some hard-to-predict ways that allow attackers to break into the system.

Random Oracle Model

To analyze a protocol using the Random Oracle Model (ROM), one replaces a real-world cryptographic hash function by a black-box that when queried outputs a random bit-string, subject to the restriction that it always outputs the same value on the same input. Having made this replacement, one then gives a reductionist security argument. The right way to view a proof of security in ROM is as a proof of security against a restricted class of adversaries that do not care if the hash function really is a black-box. This limitation has been pointed out by Canetti [19].

Standard Models

This is the preferred approach of modern, mathematical cryptography. Here, one shows with mathematical rigor that any attacker who can break the cryptosystem can be transformed into an efficient algorithm to solve the underlying well-studied problem that is widely believed to be very hard. In other words, if the "hardness assumption" is correct as presumed, the cryptosystem is secure.

This approach is about the best cryptographers can do. If security can be proved in this way, then we essentially rule out all possible attacks, even those "we have not yet imagined".

5.3 Security Analysis of the Group-Key Management Module

In this section, we adopt the Random Oracle Model and sketch proofs of two key properties of the Group-Key Management module, namely, robustness and unpredictability of the secret sharing scheme of Enclaves.

Theorem 4 (Robustness) *In the random oracle model, a dishonest leader cannot forge, with a non negligible probability, a valid proof for a non valid share.*

Proof sketch: Let s_i be the share provided by leader L_i and (n_i, M_i, u, v, z) be the corresponding correctness proof. s_i, u, v and z should then satisfy the following equations:

$$g^z = u\ {g_i}^c \tag{5.4}$$

$$\tilde{g}^z = v\ {s_i}^c \tag{5.5}$$

where $c = H_2(g, g_i, u, \tilde{g}, s_i, v)$. Equation (5.4) yields $u \in G$, since ${g_i}^c$ and g^z are both in G (closure of G under multiplication). The latter implies that it exists $\gamma \in \mathbb{Z}_q$ such that $u = g^\gamma$. Equation (5.4) gives: $g^z = g^\gamma g^{cx_i}$, which implies: $z = \gamma + cx_i$. Now Equation (5.5) becomes:

$$\begin{aligned} \tilde{g}^z = v\ {s_i}^c \iff \tilde{g}^{(\gamma + cx_i)} &= v\ {s_i}^c \\ \iff \tilde{g}^\gamma\ v^{-1} &= (\tilde{g}^{-x_i}\ s_i)^c \end{aligned}$$

This yields two possible cases:

1. $s_i = \tilde{g}^{x_i}$. In this case, the share is correct. $v = \tilde{g}^{\gamma}$ and for all $c \in \mathbb{Z}_q$ the verifier equations trivially hold.

2. $s_i \neq \tilde{g}^{x_i}$. In this case we must have

$$c = \log_{(\tilde{g}^{-x_i}\ s_i)}(\tilde{g}^{\gamma}\ v^{-1}). \tag{5.6}$$

From equation 5.6, one can see that, if s_i is not a valid share, once the triplet (s_i, u, v) is chosen, then there exists a unique $c \in \mathbb{Z}_q$ that satisfies the verifier equations. In the random oracle model, the hash function H_2 is assumed to be perfectly random. Therefore, the probability that $H_2(g, g_i, u, \tilde{g}, s_i, v)$ equals c, once (s_i, u, v) fixed, is $\frac{1}{q}$. On the other hand, if the attacker performs an adaptively chosen message attack by querying an oracle $\mathcal{N}$ times, the probability for the attacker to find a triplet (s_i, u, v), such that $c = H_2(g, g_i, u, \tilde{g}, s_i, v)$, is $\mathcal{P}_{Success} = 1 - (1 - \frac{1}{q})^{\mathcal{N}} \approx \frac{\mathcal{N}}{q}$ for large q and $\mathcal{N}$. Now if k is the number of bits in the binary representation of q, then $\mathcal{P}_{Success} \leq \frac{\mathcal{N}}{2^k}$. Since a computationally bounded leader can only try a polynomial number of triplets, then when k is large, the probability of success is negligible ($\mathcal{P}_{Success} = \frac{\mathcal{N}}{2^k} \ll 1$).

Theorem 5 (Unpredictability) *An attacker that corrupts up to f leaders cannot, with a non negligible probability, learn the secret group key $\tilde{g}^x$.*

This has been proved by Cachin, Kursawe, and Shoup [17] and relies on both:

- The perfect cryptography assumption:

$$S(s_{i_{f+1}} \mid s_{i_1}, s_{i_2}, \cdots, s_{i_j}) = S(s_{i_{f+1}}) \quad \forall j \leq f.$$

where S denotes the information entropy function.

- The Computational Diffie-Hellman assumption [30], which states that there is no polynomial time probabilistic algorithm that computes $s_i = \tilde{g}^{x_i}$ given g, $\tilde{g}$, and $g_i = g^{x_i}$, with a non-negligible probability of error.

In a nutshell, the knowledge of up to f shares does not give the attacker any statistical privilege to predict extra valid shares. Therefore, the data a computationally bounded attacker might have access to, is not sufficient to reconstruct the group key in polynomial time and with a non-negligible probability of error. A more accurate proof of a similar theorem has been devised by Cachin *et al.* in [18].

5.4 Concluding Remarks

In this chapter, we have studied the architecture and correctness of the Group-Key Management module of Enclaves. We have also reviewed some of the state-of-the-art approaches and models used in the verification of similar provably secure schemes.

The main purpose of our verification was to obtain as much precise results as possible about the security of the Group-Key Management module, rather then asymptotic results about families of cryptographic functions. Therefore, we have chosen the concrete approach, where we "quantify" the statistical advantage of an Intruder defeating a scheme. Also, and in line with our perfect cryptography assumption, we have considered cryptographic functions as black-box constructs by adopting the Random Oracle Model.

Given these assumptions, we proved the *robustness* property of the Group-Key Management module. We have also pointed out for a way to prove the *unpredictability* property, already derived in a similar context by Cachin *et al.* [18].

This work could be, however, improved by using the standard model, instead of the Random Oracle Model, thereby covering a much wider range of possible attacks.

Chapter 6

Conclusion and Future Work

In this thesis, we have presented the formal modeling and verification of the Intrusion-Tolerant Enclaves protocol. We experiment with an adaptive combination of techniques namely model checking, theorem proving and mathematical analysis. Our choices of the used techniques and tools were based on the nature of the correctness arguments in each module of the protocol, on the environment assumptions and the easiness of performing the verification.

We have specified the Authentication module of Enclaves in Murphi and checked two invariants of the module reachable states, namely *clients* and *leaders proper authentication*. We have presented the experimental results of the invariants checking for different sizes of the protocol. Murphi passed the check for small instances, but as we started increasing the size parameters, the tool was unable to terminate the model checking procedure.

We have used the PVS theorem prover to specify and prove the Byzantine Agreement module of Enclaves. For instance, we have proved the protocol to satisfy its requirements of *Termination*, *Integrity* and *Proper Agreement.* Given the high level of expressiveness of the Timed-Automata formalism, as well as the rich datatype package of PVS, we succeeded to formalize the Byzantine agreement module for any number of leaders, in a way that reasonably captures the many subtleties on which the correctness arguments of Enclaves rely.

Finally, we sketched a manual proof of the *robustness* and *unpredictability* properties of the secret sharing scheme of Enclaves using the Random Oracle Model (ROM). In this model, real-world cryptographic hash functions are idealistically assimilated to *black-box* constructs that when queried, output *random* bit strings. This idealization allows only proofs of security against a restricted class of attacks, ones that do not exploit whether a hash function has a particular structure or not.

We believe to have achieved a promising success in verifying a complex protocol such as Enclaves. Our results could be improved though. For instance, the feasibility of model checking, is always limited to instances with a finite number of states, which may, in some cases, prevent from discovering security flaws in realistic implementations of the protocols. To improve this, one could use rank functions with theorem proving [94]. We believe that using rank functions is a very efficient way to mechanically prove authentication properties for any generic size of protocols. We are considering this approach among our future work plans.

Aside from that, we have proved the consistency of group membership only when prospective clients ask to join the group. We still have not yet considered the case where members leave the group, for instance. This is also among our future work.

Furthermore, we have used the Random Oracle Model in our analysis of the secret sharing scheme of Enclaves. The Random Oracle Model allows only for reasoning about a limited class of attacks. The standard model is an alternative. This latter comprehends all possible cryptographic primitives and adversarial models. If we can prove security in the standard model, then we essentially rule out all possible attacks, even those "not imagined yet". It would be interesting then, to consider this avenue in future research. Also, another attractive research direction would be to conduct mathematics related proofs mechanically in a theorem prover, thereby allowing us to prove all three parts of Enclaves in PVS for instance. This will require the elaboration of a set of specialized theories (e.g., probabilities), already explored in HOL [52], but not yet in PVS.

The current specification can be extended even further by widening the Byzantine faults capabilities and by introducing the joint cryptographic layers that have been abstracted away. Also results about an upper bound on Agreement establishment delays can be further investigated.

The complete Murphi and PVS source codes for our work can be found at `http://hvg.ece.concordia.ca/Research/CRYPTO/Enclaves.html`.

Bibliography

[1] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 36–47, Zurich, Switzerland, April 1997. ACM Press.

[2] R. Accorsi, D. Basin, and L. Viganò. Towards an Awareness-Based Semantics for Security Protocol Analysis. *Electronic Notes in Theoretical Computer Science*, 55(1):9–28, 2001.

[3] K. Adi and M. Debbabi. Abstract Interpretation for Proving Secrecy Properties in Security Protocols. *Electronic Notes in Theoretical Computer Science*, 55(1):29–54, 2001.

[4] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J.

Schultz, J. Stanton, and G. Tsudik. Secure Group Communication in Asynchronous Networks with Failures: Integration and Experiments. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 330–343, Taipei, Taiwan, April 2000. IEEE Computer Society.

[6] M. Archer. Proving Correctness of the Basic TESLA Multicast Stream Authentication Protocol with TAME. Presented at the Second Workshop on Issues in the Theory of Security, Portland, Oregon, USA, January 2002. Available online at http://chacs.nrl.navy.mil/5540/personnel/archer.html.

[7] M. Archer, C.L. Heitmeyer, and E. Riccobene. Proving Invariants of I/O Automata with TAME. *Automated Software Engineering*, 9(3):201–232, August 2002.

[8] M. Schunter B. Pfitzmann and M. Waidner. Cryptographic Security of Reactive Systems. In *Proceedings of the Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*, Royal Holloway, London, UK, December 2000. Elsevier Science Publishers.

[9] M. Bellare and P. Rogaway. Random Oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 1993. ACM Press.

[10] C.H. Bennett, G. Brassard, R. Jozsa, D. Mayers, A. Peres, B. Schumacher, and W.K. Wootters. Reduction of Quantum Entropy by Reversible Extraction of Classical Information. *Journal of Modern Optics, Special issue on Quantum Communication and Cryptography*, 41(12):2445–2454, December 1994.

[11] E. Biham, D. Boneh, and O. Reingold. Breaking Generalized Diffie-Hellman Modulo a Composite is no easier than Factoring. *Information Processing Letters*, 70:83–87, 1999.

[12] R.S. Boyer and J.S. Moore. *A Computational Logic*. ACM mono-graph series. Academic Press, 1979.

[13] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[14] A. Z. Broder and D. Dolev. Flipping Coins in Many Pockets (Byzantine Agreement on Uniformly Random Values). In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 157–170, Singer Island, Florida, USA, October 1984. IEEE Computer Society Press.

[15] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[16] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[17] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. In *Proceedings of the 19th Annual Symposium on Principles of Distribtued Computing*, Portland, Oregon, USA, July 2000. ACM Press.

[18] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. In *Cryptology ePrint Archive, http://eprint.iacr.org*, 2000. Revised version.

[19] R. Canetti. Towards Realizing Random Oracles: Hash Functions That Hide All Partial Information. In *Advances in Cryptology*, volume 1294 of *Lecture Notes in Computer Science*, pages 455–469. Springer Verlag, 1997.

[20] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Report MIT/LCS/TM-590, MIT Laboratory for Computer Science, June 1999.

[21] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ACM Special Issue, pages 173–186, New Orleans, Louisiana, USA, February 1999. ACM Press.

[22] Cisco Systems Inc. Cisco Secure IDS (formerly NetRanger). June, 2003.

[23] D. Chaum and T. Pedersen. Wallet Databases with Observers. In *Advances in Cryptology*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer Verlag, 1992.

[24] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *Proceedings of the IEEE Symposium on the Foundations of Computer Science*, pages 383–395, Portland, Oregon, USA, 1985. IEEE Computer Society.

[25] J. Clark and J. Jacob. A Survey of Authentication Protocols Literature: Version 1.0. Department of Computer Science, University of York, UK, 1997.

[26] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State Space Reduction Using Partial Order Techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[27] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[28] M. Crosbie. Applying Genetic Programming to Intrusion Detection. In *Proceedings of the American Association for Artificial Intelligence Fall Symposium on Genetic Programming*, Cambridge, Maryland, USA, November 1995.

[29] Y. Desmedt and Y. Frankel. Shared Generation of Authenticators and Signatures. In *Advances in Cryptology*, volume 576 of *Lecture Notes in Computer Science*, pages 457–469. Springer Verlag, 1991.

[30] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[31] D. Dill, A. Drexler, A. J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 11–14, Cambridge, Maryland, USA, October 1992. IEEE Computer Society.

[32] D. Dolev and A.C. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 29(12):198–208, March 1983.

[33] B. Dutertre, V. Crettaz, and V. Stavridou. Intrusion-Tolerant Enclaves. In *Proceedings of the IEEE International Symposium on Security and Privacy*, Oakland, California, USA, May 2002. IEEE Computer Society.

[34] B. Dutertre, H. Saidi, and V. Stavridou. Intrusion-Tolerant Group Management in Enclaves. In *International Conference on Dependable Systems and Networks*, pages 203–212, Göteborg, Sweden, July 2001. IEEE Computer Society.

[35] E.A. Emerson. Symmetry and Model Checking. In *Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 463–478. Springer Verlag, 1993.

[36] R. Fagin and J.Y. Halpern. Belief, Awareness, and Limited Reasoning. *Artificial Intelligence*, 34(1):39–76, 1987.

[37] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning About Knowledge.* MIT Press, 1995.

[38] P. Feldman. A Practical Scheme for Non-interactive Verifiable Secret Sharing. In *Proceedings of the IEEE Symposium on the Foundation of Computer Science*, pages 427–437, Los Angeles, California, USA, 1987. IEEE Computer Society.

[39] M. Fischer, N. Lynch, and S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[40] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust Threshold DSS Signatures. In *Advances in Cryptology*, volume 1070 of *Lecture Notes in Computer Science*, pages 354–371. Springer Verlag, 1996.

[41] S.L. Gerhart, D.R. Musser, D.H. Thompson, D.A. Baker, R.L. Bates, R.W. Erickson, R.L. London, D.G. Taylor, and D.S. Wile. An Overview of AFFIRM: A Specification and Verification System. In *Proceedings of the ACM Conference on Performance of the System R Access Path Selection Mechanism*, pages 343–347, Tokyo, Japan, 1980.

[42] D. Gollmann. What Do we Mean by Entity Authentication? In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 46–54, Oakland, California, USA, May 1996. IEEE Computer Society.

[43] D. Gollmann. On the Verification of Cryptographic Protocols - A Tale of Two Committees. In *Proceedings of the Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*, Royal Holloway, London, UK, December 2000. Elsevier Science Publishers.

[44] L. Gong. Enclaves: Enabling Secure Collaboration over the Internet. *IEEE Journal of Selected Areas in Communications*, 15(3):567–575, April 1997.

[45] L. Gong, R. Needham, and R. Yahalom. Reasoning about Belief in Cryptographic Protocols. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 234–248, Oakland, California, USA, May 1990. IEEE Computer Society.

[46] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge, UK, 1993.

[47] J. Haines, R. Lippmann, D. Fried, E. Tran, S. Boswell, and M. Zissman. 1999 DARPA Intrusion Detection System Evaluation: Design and Procedures. Technical report, MIT Lincoln Laboratory, 2001.

[48] J. Haines, L. Rossey, R. Lippmann, and R. Cunnigham. Extending the 1999 Evaluation. In *Proceedings of the Second DARPA Information Survivability*

Conference and Exposition, Anaheim, California, USA, June 2001. IEEE Computer Society.

[49] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. In *Proceedings of the Seventh Symposium on Logics in Computer Science*, Santa-Cruz, California, June 1992. IEEE Computer Society.

[50] J. Hickey, N. Lynch, and R. van Renesse. Specifications and Proofs for Ensemble Layers. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Copmuter Science*, pages 119–133. Springer Verlag, 1999.

[51] A.J. Hu, R. Li, X. Shi, and S.T. Vuong. Model-Checking a Secure Group Communication Protocol: A Case Study. In *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*, pages 5–8, Beijing, China, October 1999. Kluwer Academics.

[52] J. Hurd. A Formal Approach to Probabilistic Termination. In *Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 230–245. Springer Verlag, 2002.

[53] IEEE Standards. *Wired Equivalent Privacy for Wireless LANs (IEEE 802.11-1997)*, 1997.

[54] International Organization for Standardization/International Electrotechnical Commission. *Information Technology, Security Techniques, Entity Authentication Mechnismes Part1: General Model (ISO/ESC 9798)*, 1997.

[55] C.N. Ip and D. Dill. Better Verification through Symmetry. In *Proceedings of the International Conference on Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Ontario, Canada, April 1993. North-Holland.

[56] C.N. Ip and D. Dill. Efficient Verification of Symmetric Concurrent Systems. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, Cambridge, Maryland, USA, October 1993. IEEE Computer Society.

[57] C.N. Ip and D. Dill. State Reduction Using Reversible Rules. In *Procedings of the 33rd Design Automation Conference*, pages 564–567, Las Vegas, Nevada, USA, June 1996. ACM Press.

[58] C.N. Ip and D. Dill. Verifying Systems with Replicated Components in Murphi. In *Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 147–158. Springer Verlag, 1996.

[59] M. Kwiatkowska and G. Norman. Verifying Randomized Byzantine Agreement. In *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems*, volume 2529 of *Lecture Notes in Computer*

Science, pages 194–209. Springer Verlag, 2002.

[60] L. Lamport, R.E. Shostak, and M.C. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[61] P. Lincoln and J.M. Rushby. A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model. In *Proceedings of the Twenty-Third Annual International Symposium on Fault-Tolerant Computing*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[62] R. Locasso, J. Scheid, D.V. Schorre, and P.R. Eggert. The Ina Jo Reference Manual. Technical Report TM-(L)-6021/001/000, System Development Corporation, Santa Monica, California, 1980.

[63] G. Lowe. A Hierarchy of Authentication Specifications. In *Proceedings of The 10th Computer Security Foundations Workshop*, pages 31–44, Rockport, Massachusetts, USA, June 1997. IEEE Computer Society.

[64] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[65] N. Lynch and M. Tuttle. An introduction to input/output automata. *Centrum voor Wiskunde en Informatica Quarterly Journal*, 2(3):219–246, 1989.

[66] U. Maurer. Towards Proving the Equivalence of Breaking the Diffie-Hellman Protocol and Computing Discrete Logarithms. In *Advances in Cryptology*,

volume 839 of *Lecture Notes in Computer Science*, pages 271–281. Springer Verlag, 1994.

[67] K. McCurley. A Key Distribution System Equivalent to Factoring. *Journal of Cryptology*, 1:95–105, 1988.

[68] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academics, 1993.

[69] C. Meadows. Formal Verification of Cryptographic Protocols: A Survey. In *Advances in Cryptology*, volume 917 of *Lecture Notes in Computer Science*, pages 133–150. Springer Verlag, 1994.

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

[71] J. Millen, S.C. Clark, and S.B. Freedman. The Interrogator: Protocol Security Analysis. *IEEE Transactions on Software Engineering*, 13(2), February 1987.

[72] MIT Lincoln Laboratory - DARPA Intrusion Detection Evaluation Publications. http://www.ll.mit.edu/IST/ideval/pubs/pubs_index.html (1998-2001).

[73] J.C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 141–153, Oakland, California, USA, May 1997. IEEE Computer Society.

[74] J.C. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In *Proceedings of the Seventh USENIX Security Symposium*, pages 201–215, San Antonio, Texas, USA, January 1998.

[75] National Bureau of Standards. *Data Encryption Standard, (FIPS 46)*, 1977.

[76] National Bureau of Standards. *Advanced Encryption Standard (FIPS 197)*, 2001.

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

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

[79] P. Neumann and P. Porras. Experience with EMERALD to Date. In *Proceedings of the first USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, Santa Clara, California, USA, April 1999.

[80] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Verlag, 1992.

[81] S. Park and D. Dill. An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order). In *Proceedings of the Seventh Annual ACM*

Symposium on Parallel Algorithms and Architectures, pages 34–41, Santa Barbara, California, USA, July 1995. ACM Press.

[82] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

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

[84] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the Seventh USENIX Security Symposium*, pages 31–52, San Antonio, Texas, USA, January 1998.

[85] T. Pedersen. Non-interactive and Information-theoretic Secure Verifiable Secret Sharing. In *Advances in Cryptology*, volume 1070 of *Lecture Notes in Computer Science*, pages 129–140. Springer Verlag, 1992.

[86] A. Perrig, R. Canetti, D. Tygar, and D. Song. Efficient Authentication and Signing of Multicast Streams over Lossy Channels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 56–73, Berkeley, California, USA, May 2000. IEEE Computer Society.

[87] D. Pointcheval and S. Vaudenay. On Provable Security for Digital Signature Algorithms. Technical report, Ecole Normale Supérieure, Paris, France, Decembre 1996.

[88] R.D. Prisco, B.W. Lampson, and N.A. Lynch. Revisiting the PAXOS Algorithm. In *Distributed Algorithms*, volume 1320 of *Lecture Notes in Computer Science*, pages 111–125. Springer Verlag, 1997.

[89] M. Reiter. The Rampart Toolkit for Building High-Integrity Services. In *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110. Springer Verlag, 1995.

[90] R.L. Rivest, A. Shamir, and L.M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[91] O. Rodeh, K. Birman, and D. Dolev. The Architecture and Performance of the Security Protocols in the Ensemble Group Communication System. *ACM Transactions on Information Systems and Security*, 4(3):289–319, 2001.

[92] RSA Security. *RSA Cryptography Standard (PKCS 1)*, 2002.

[93] P. Ryan and S. Schneider. *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison-Wesley, 2000.

[94] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22:612–613, 1979.

[95] V. Shoup. Practical Threshold Signatures. In *Advances in Cryptology*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer Verlag, 2000.

[96] S.R. Snapp, S.E. Smaha, T. Grance, and D.M. Teal. The DIDS (Distributed Intrusion Detection System) Prototype. In *Proceedings of the USENIX Summer Technical Conference*, pages 227–233, San Antonio, Texas, USA, June 1992.

[97] U. Stern and D. Dill. Improved Probabilistic Verification by Hash Compaction. In *Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 206–224. Springer Verlag, 1995.

[98] M. J. Toussaint. Deriving the Complete Knowledge of Participants in Cryptographic Protocols. In *Advances in Cryptology*, volume 576 of *Lecture Notes Computer Science*, pages 24–43. Springer Verlag, 1991.

[99] H. van der Schoot and H. Ural. An Improvement in Partial-Order Verification. *Software Testing, Verification, and Reliability*, 8(2):83–102, 1998.

[100] R. P. Venkat. An Axiomatic Basis for Trust in Distributed Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 204–211, Washington DC, USA, April 1988. IEEE Computer Society.

[101] P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer Verlag, 1993.

[102] J. Zhou and D. Gollmann. A Fair Non-repudiation Protocol. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 46–54, Oakland, California, USA, May 1996. IEEE Computer Society.

[103] U.S. Air Force Information Warfare Center and Trident Systems. ASIM (Automated Security Incident Measurement). http://www.access.gpo.gov (1995).