Technical Report

UCAM-CL-TR-479
ISSN 1476-2986

Number 479

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

The interaction between fault tolerance and security

Geraint Price

December 1999

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted
June 1999 by the author for the degree of Doctor of
Philosophy to the University of Cambridge, Wolfson College.

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

The Interaction Between Fault Tolerance and Security

Geraint Price

Wolfson College
University of Cambridge

June 1999

A dissertation submitted for the degree of
Doctor of Philosophy

Summary

This dissertation studies the effects on system design when including fault tolerance design principles within security services.

We start by looking at the changes made to the trust model within protocol design, and how moving away from trusted server design principles affects the structure of the protocol. Taking the primary results from this work, we move on to study how control in protocol execution can be used to increase assurances in the actions of legitimate participants. We study some examples, defining two new classes of attack, and note that by increasing client control in areas of protocol execution, it is possible to overcome certain vulnerabilities.

We then look at different models in fault tolerance, and how their adoption into a secure environment can change the design principles and assumptions made when applying the models.

We next look at the application of timing checks in protocols. There are some classes of timing attack that are difficult to thwart using existing techniques, because of the inherent unreliability of networked communication. We develop a method of converting the Quality of Service mechanisms built into ATM networks in order to achieve another layer of protection against timing attacks.

We then study the use of primary-backup mechanisms within server design, as previous work on server replication in security centres on the use of the state machine approach for replication, which provides a higher degree of assurance in system design, but adds complexity.

We then provide a design for a server to reliably and securely store objects across a loosely coupled, distributed environment. The main goal behind this design was to realise the ability for a client to exert control over the fault tolerance inherent in the service.

The main conclusions we draw from our research are that fault tolerance has a wider application within security than current practices, which are primarily based on replicating servers, and clients can exert control over the protocols and mechanisms to achieve resilience against differing classes of attack. We promote some new ideas on how, by challenging the prevailing model for client-server architectures in a secure environment, legitimate clients can have greater control over the services they use. We believe this to be a useful goal, given that the client stands to lose if the security of the server is undermined.

Acknowledgements

I would like to thank Roger Needham for being my supervisor and for his patience. I would also like to thank Bruno Crispo, Steve Hand, Václav Matyáš and Kan Zhang for various comments on early sections of this work. I would also like to thank Bruce Christianson for helping me improve on the presentation of this dissertation.

I would like to thank my parents for both their emotional and financial support, and my sisters Mair and Ann for their humour and emotional support.

This work was supported by a studentship from the Engineering and Physical Sciences Research Council.

Declarations

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

This dissertation in not substantially the same as any other that I have submitted for a degree, diploma, or other qualification at any other university.

No part of this dissertation has already been, or is currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

I Taid,
Mor falch o weld fi'n dechrau.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

In this chapter we give a brief outline of the dissertation, giving a small overview of each chapter and a structure of how the work fits together.

1.2 Synopsis

In this dissertation we aim to study the use of principles used in fault tolerance within secure services. There is little work done in the field at present. The work that has been done so far concentrates on the use of replication to increase the resilience and availability of servers.

We aim to take a step back from the notions used in the existing work, in order to see how the basic principle behind fault tolerance of using checks and control of flow can help improve the security of protocols and services, and also increase client control in some situations. We believe both of these goals to be of potential use in many scenarios within security. The threat model used in previous work maintains the assumption of a trusted server, but that clients are allowed to be malicious. Previous work uses fault tolerance predominantly as an internal feature of the server to improve the assurances given to the client of the server's reliability, and is not used to challenge the client's need to rely on the server. We try and turn this notion on its head, allowing the client to define the trust parameters in the rest of the system, and in doing so, providing them with some measure of control over the system's ability to tolerate failure.

1.3 Structure

We now give a brief overview of the structure of our work.

Our work breaks down into three broad areas, and how they fit the structure of this dissertation is shown in figure 1.1.

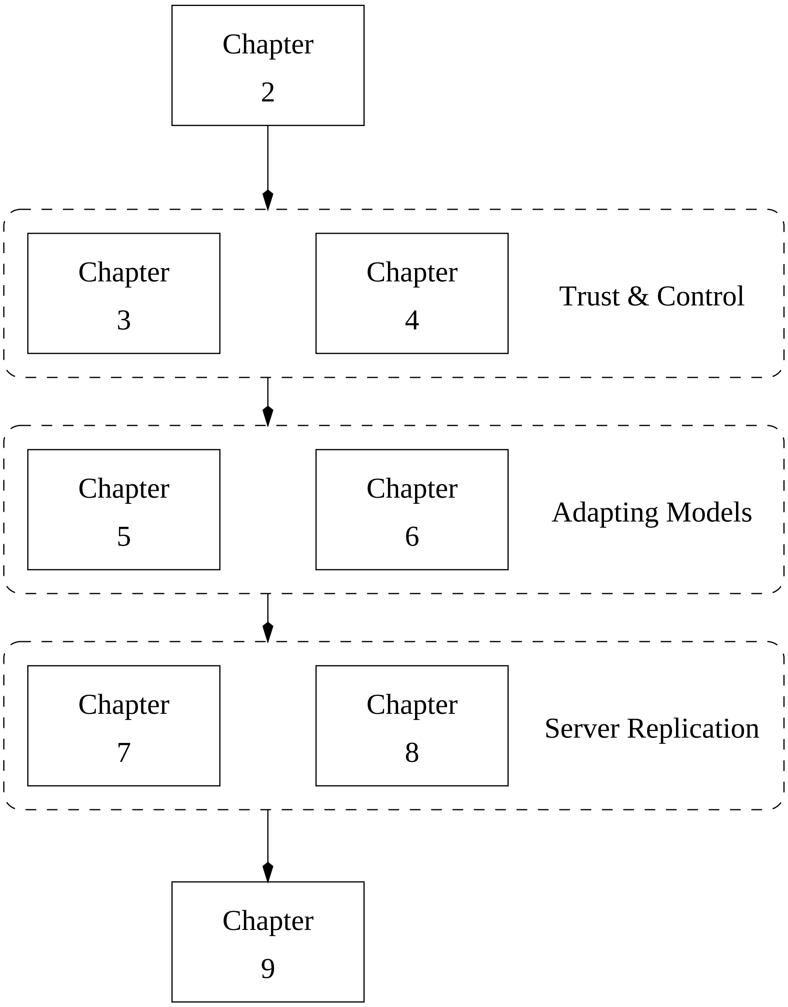

Figure 1.1: Overview of the structure of the dissertation

The first area concerns itself with the adoption of the notion of fault tolerance to security, both in terms of strictly adhering to fault tolerant design principles, and also integrating the notion of checks and control that is prevalent in the fault tolerance literature. These areas are encapsulated by the work carried out in chapters three and four respectively.

The second area concentrates on the adaption of models other than those used purely for server replication, as most of the current work done on integrating fault tolerance and security focus mainly on server replication. We aim to see how this changes our design principles, and this covers the work done in chapters five and six.

The third area of research looks at differing approaches to server replication. Current work focuses on state machine replication. We approach server replication from two different angles in chapters seven and eight to look at different methods to achieving replication securely under differing threat models.

In the next section we provide a fuller overview of each chapter.

1.4 Summary

In the second chapter we give a brief overview of some of the work that has both driven our research and inspired some of our discoveries. The key points within fault tolerance are the developments of the notions of faults and failures, the use of replication in services, and reliability measurement. The areas covered within the security literature cover the existing work that studies the use of fault tolerance, and also the notion of denial of service.

Our third chapter studies the effect that using fault tolerant principles has on the notions of belief and trust that are prevalent in the protocol design literature. We start by looking at what constitutes a fault within a secure protocol, taking the definitions used in the fault tolerance literature and developing equivalents for security.

To highlight our new definitions, we take the BAN logic for protocol analysis. Their work relies heavily on the notion of trusted principals. We relax this model by use of the classic N-model redundancy technique. We modify the logic to accommodate this design in order to gain some insight into what new design principles need to be incorporated for protocols to still meet their security goals in the different environment.

In the fourth chapter, we draw the conclusion that the application of checks and control that is inherent in fault tolerance can be embodied in the controlling structure of a secure protocol. We study three variants of how control of the protocol's execution, and control of data flow in a protocol can provide higher levels of assurance toward the protocol achieving its goals.

Our second example in the chapter provides us with evidence that, even if protocols are currently designed to withstand certain classes of attack, then by allowing the participant concerned with the attack to achieve a greater degree of control over the protocol, we can increase the assurance we have in the protocol.

Our fifth chapter studies how models from the fault tolerant literature – other than those currently used for server replication – adapt to use in a secure environment.

The existing work on the use of fault tolerance in secure services centres on the use of the state machine approach to server replication. We take three other design principles within fault tolerance and carry them over for use in security. Our goal is to see if changes need to be made to these three models, and to try and understand any underlying principles we can derive from the resulting changes.

In chapter six, we take the notion of timing checks and apply them in a secure environment. Timing and replay attacks are prevalent in the literature, and there already exist methods to thwart such attacks, but there are some classes of timing attacks that can still be carried out because of the inherent unreliability of networked communication.

We develop a technique that achieves another layer of protection against timing attacks by utilising the quality of service mechanisms built into ATM networks. We take an existing taxonomy of replay attacks and analyse how well our method fares in defending against each case. We do not regard our mechanism as a replacement for existing methods of defeating such attacks, but notice that it complements existing techniques. As a by-product of its behaviour, it can also provide a tool for detecting potential denial of service attacks.

Chapter seven looks at the use of primary-backup strategy of server replication. As stated above, the majority of work done to date on fault tolerance's integration in security uses the state machine approach. Although more robust, the state machine approach involves a higher degree of complexity in the design.

We look at some instances where clients are willing to trust some replicas more than others, and also where server replication can be used to increase the availability of the service in the case of a benign crash, or a denial of service attack, where the primary threat model is not concerned with the malicious takeover of the server.

In chapter eight we provide a design for a server to reliably and securely store objects across a loosely coupled distributed environment. The main goal behind this design, was to demonstrate the ability for the client to exert control over the fault tolerance inherent in a service.

The design principle we use is that of the state machine approach, but differs from current work in this field which centres solely on the method's use as a means of increasing the assurance that the service is trustworthy.

Chapter nine provides an overview of the results drawn together in the remainder of the dissertation, along with an outline of some possible avenues of research which we highlight from insights gained while carrying out our work.

Chapter 2

Background

2.1 Overview

In this chapter, we review current work in security that has been of relevance to our research, and also cover work done within fault tolerance which has been an influence on our work.

We have arranged our discussion of current work into sections dealing with particular themes.

2.2 Fault Tolerance

Fault Tolerance has been part of the computing community for quite a long time, with quite a wide an varied set of literature. To clarify the building of our understanding of fault tolerance, then we divide our discussion into three sections. Firstly we cover some early work done in fault tolerance, then look at some work done on group based communication, and conclude with a look at a variety of loosely connected work within the field.

2.2.1 Core principles

Newcastle University developed some of the very early work in fault tolerance, through a group led primarily by Randell and Anderson. Their work covers much of the early aspects of fault tolerance [AL81, ALS79, ALS78, AL82, MR77, CAR83, DR86].

Anderson and Lee [AL82] set out to define fault tolerance terminology, focusing on the difference between a *fault*, *error* and *failure*. These terms are given strict definition in order to avoid their use interchangeably. They also break down fault tolerance into four constituent phases of *fault detection*, *damage assessment*, *error recovery* and *fault treatment*. Further on in this dissertation, we discuss how this terminology can be viewed from within a security context, and the effect this can have on our design principles.

In an overview of the subject [AL81], they bring together some interesting concepts which is an expansion of their work with Shrivastava [ALS79]. In [AL81] the discussion is broad and wide-ranging. Their description of fault tolerance in the context of state transitions [AL81, pages 46–50] is interesting, showing how the internal state of the system can be viewed as valid or erroneous, and that if an erroneous state goes unchecked, then a failure of the system can arise.

A section of their work [AL81, pages 72–77] overviews the principles of Mean Time Between Failure (MTBF) and Mean Time To Repair (MTTR), which are two core principles used in the study of fault tolerance in hardware systems. Although there is a small body of work done by others [LBF+93, Mea94, Mea95] covering the application of this work to software and security, we believe these principles to be of little relevance to our work, and elaborate on our arguments when covering their work in section 2.3.

In their section on recovery of distributed systems [AL81, pages 208–224] they introduce the difference between recovery for systems where process are *competing* and *cooperating*. They explain the difference as one of communication between the processes, and they note difficulty that the *domino effect* can have on the cooperating model.

Anderson et al. in [ALS79, pages 179–190] and [ALS78] discuss the use of recovery in multilevel systems. The main feature of their work that is of interest to us is the definition of *inclusive* and *disjoint* recovery schemes. These two schemes arise from the difference that arises from implementing recovery across an interface, where the interface itself is an extension to a lower level interface, and both interfaces are open to the calling module. The difference lies in the fact that recovery is co-ordinated across the interface without any need for intervention in inclusive, while the extension has to independently carry out recovery in disjoint.

Merlin and Randell [MR77] discuss the difficulty in providing state restoration in a distributed system. Their model for recovery involves backward error recovery, with information for interaction connections built in what they term an “Occurrence Graph Model” which is an extension of Petri-nets.

Campbell et al. [CAR83] make the interesting observation, that fault tolerance in an asynchronous system is best implemented by using atomic actions, as this provides a clean interface that works for the recovery irrespective of the timing in the communication model.

Dobson and Randell [DR86] discuss the effect of placing of the security perimeter in a processing environment, and note that the different security boundaries vary according to whose view of the boundary we are interested in.

2.2.2 Cornell

A significant portion of the work in software replication has come out of Cornell University, and there are three aspects of their work of interest to us. The first is with regard to the work done on group and multicast communication, the second is the replication of services using the state-machine approach, and the third is the recent movement to try and apply fault tolerant techniques in the advent of mobile agents.

Group Communication

There has been work done in many places on the design and use of group communication primitives. The most comprehensive and long running study has come out of Cornell.

Their early work studied the bounds under which group consensus could be achieved. This work was an extension of the seminal formulation of the problem termed *Byzantine Agreement* [LSP82]. Their work [BT85, BT83] demonstrated bounds for reaching consensus in an asynchronous environment under both benign and malicious failure.

The extension from their first results was the *Isis* toolkit [BSS91, RBG92]. The toolkit allows process groups to be formed, and then a number of primitives are available for communication within the group. The main primitives developed for group communication are a CBCAST protocol, which gives all broadcasts within the group a causal order. On top of this was added the ABCAST protocol, which allows for the causal order primitive to be upgraded to an atomic ordering. These mechanisms allow for a *virtually synchronous* group communication mechanism to be implemented directly below the application layer, making it easier for the application programmer. Their work was built on the assumption of benign failures only.

Their work in *Isis* was extended to *Horus* [vRBM96], which allowed for a more flexible approach to the type of the services presented to the application.

Another extension of the work done initially in *Isis*, but developed separately from *Horus*, was that of *Rampart.* This work was carried out by Reiter [Rei96a, Rei94b, Rei94a] after he left Cornell, but the connection with the history of *Isis* is clearly visible in the protocol structure. The main advancement *Rampart* achieved over *Isis* was the ability to tolerate malicious failures of a given percentage of servers in a group.

State Machine Approach

Schneider [Sch90] proposes the use of the State Machine Approach in order to replicate servers. The main goal is the ability to replicate a server in order that it is resilient against the failure of a significant portion of the server without compromising the service provided. The notion behind the design is to have several instances of the server all with equal ability, communicating to decide on the the next global action.

It shares many characteristics with other work done by Schneider regarding fail-stop processors [SS83, Sch84]. This work differs from the work on state machines by the virtue that a set of processors are able to detect that they are about to fail, and halt before any fault is visible to the outside world. In order to achieve this, they require an underlying replication mechanism, such that correct processes in a group can determine that there is a failure in the group, and gracefully disrupt the computation accordingly.

Although the work on the State Machine Approach was published after the work on fail-stop processors, it is possible to see the underlying principle is prevalent in the earlier work.

Secure Agentry

A growing area of interest in distributed systems is the use of mobile agents. The algorithms they represent are calculated on machines not under the control of the source of the code, but the originator still wants to be assured of the the integrity of the results. To circumvent this problem, Schneider [Sch97, JvRS96] looks at ways of implementing fault tolerance into the agent structure. The protocols he derives are based upon having the agents replicate themselves for each step of the computation, producing separate results on different processors. These results are then used in a voting algorithm for each of the replicas at the next step in the computation (the assumption here is that a single agent would normally take several steps, each executed on a different machine in order to complete its task, and that each of steps is independent and is replicated autonomously by all agents).

Minsky et al. [MvRS96] describe the basic overview of their voting mechanism. The construction it uses is a secret sharing scheme, with the output at round i providing the input for the vote at round $i+1$. When using secret sharing, a minority of servers in separate rounds of the computation could collude to re-compute the secret for another round, defeating this requires updating the secret before and after a round. As an alternative, they outline a method that uses chains of authenticators.

2.2.3 Miscellany

We now overview some other work in fault tolerance which has influenced our work.

A different method of replication to the state machine approach is the primary backup approach [AD76]. This differs by having one server act as the primary point of contact, with others acting as standby. We develop upon this idea later in the dissertation for use in secure services. Work by Gifford [Gif79] is interesting in its description of using different thresholds in the read and write operations of replicated objects. The concept of weighting different replicas is also considered in our work on primary backup.

The notion of primary backup has also been used to implement a file system [MHS89]

Gong [Gon94] looks at adapting the concept of fail-stop processors for security protocols. He goes on to further the work with Syverson [GS95]. The basic premise behind the work is that if a protocol can be shown to conform to secrecy assumptions, and a malformed message arrives, then it is desirable not to send any subsequent messages in the protocol. This result is derived from not wanting to reveal any extra information to a potential attacker. A comment made by Lomas [Lom] that the correct approach should be to send a random string of the same length as the expected message, and thus not even leak the information that the valid participant knows the attack is in place. The problem with both of these scenarios is that they do not address how to recover from such a situation. In the case of Lomas's scheme, it is then impossible for one side to know who started sending the random strings in reaction to an attack.

2.3 Reliability Measurement

As noted in the section earlier by work on Anderson et al. [AL81, pages 72–77] the main metrics used in reliability measurement are Mean Time Between Failure and Mean Time To Repair. These are derived from the analysis of hardware faults, and were found to be lacking in the transition to software assessment.

Another method of describing these attributes is seen as portraying them as *reliability* and *availability* [SV97, Lit79]. Reliability is stated as the likelihood that a system will remain operational during a given period, and availability is the fraction of time that a system is operational. Littlewood [Lit79] points out that these metrics, while worthwhile in their original environments of hardware reliability, fall short of their expectations when applied to software systems. He describes other methods of trying to predict software failure by examining the relationship between the number of 'bugs' found over the working life of a piece of software, and trying to extrapolate to its future working life.

There has been some work done by Littlewood et al. [LBF$^+$93] and Meadows [Mea94, Mea95] on trying to apply the models based on reliability measurement to the security paradigm. We are not convinced that this is a direction in which such cross disciplinary research should be aimed.

In [LBF$^+$93] they argue that security faults arise from design faults which are vulnerabilities on the system. While we agree with such a definition, their motivation to study the injection of these on a statistical basis seems flawed[1]. They also speak of the difference between *accidental* and *intentional* faults. While the distinction is quite clear, there seems to be little that can be gained by making the distinction. We argue this point because an intentional fault injected into a system presents the same problem for detection as an accidental fault because the resulting events are the same when seen from the system's point of view.

In her call for a model for security analysis, Meadows [Mea94] states that any model that is built should have two attributes, firstly, it should be able to characterise attacks, and secondly that it should be able to rank attacks in order of their likelihood. She notes that the ranking of likelihood is going to be an extremely difficult problem to address. To draw an analogy, in software fault tolerance, the rate of finding 'bugs' is seen as a likely expression of future failures, but with a secure system a vulnerability can be discovered overnight and used to attack many systems. A good example of this is the dissemination – via hacker sites on the web – of shell scripts that can trigger newly discovered bugs in fielded code. This dynamic property of security flaws can compound the generation of models. Meadows [Mea95] continues by outlining a potential model for use in security. She divides security faults into three areas, the first refers to faults in the security mechanism itself, the second refers to hostile attacks, and the third refers to non-malicious human interaction of the form studied by Anderson [And94]. She concludes by stating that finding operational measures for many of these scenarios is going be difficult. We agree with her observation that measurement of the

[1] Our reasons for stating this are the same for rejecting the use of fault injection [HTI97], as stated in the last paragraph of this section.

first type of metric is already reasonably straight forward (e.g., the capacity of a covert channel, or the time taken to break a ciphertext), but these measurements provide the least insight into breaks that have to be dealt with once the system is up and running, as they tend to provide input into the design stage (i.e., use of an algorithm of an appropriate bit length depending upon the duration for which the ciphertext is expected to remain secret).

One tool for reliability measurement that we believe to be of little use in computer security is that of fault injection [HTI97]. Our position on this is deduced from the impression that any fault injection process in security offers little beyond the testing of code for bugs. While this is a valid pursuit, achieving any additional results within security is unlikely, given that any fault that is to be injected has to be conceived before hand, and it is notorious in the field of security that faults in protocols are difficult to predict.

2.4 Denial of Service

The connection between fault tolerance and denial of service is one that stems from the desired ability of a security service to continue providing functionality in the face of an attack. Given the notion that an attack on the security system can be modelled as a fault, the link between the two is clear. Unfortunately, there is little in the literature that discusses the threat of denial of service attacks. Needham [Nee94] provides comment on the threat of denial of service to a burglar alarm, discussing attacks on either the client, network or server. Gligor [Gli86] takes the view that denial of service relates to a group of authorised users being able to increase the *Maximum Waiting Time* (MWT) of a group of other authorised users. While we agree that this is one aspect of denial of service, we do not believe this to encompass all that denial of service reflects. Indeed, later on in this dissertation we introduce a denial of service attack which is not encompassed by this model, where the service itself tries to increase MWT for groups of authorised users.

He also notes the difference between *legitimate* denial of service, and other forms of denial of service. He classes legitimate denial of service as instances where service is lost due to such things as system crashes. As they tend to be temporary in nature, if there is a recovery procedure that allows a crashed system to return before the MWT, then these can be ignored by definition. He does note that when the return to service from a legitimate crash cannot be guaranteed, inclusion in the model is warranted.

Very little has been published about the study of denial of service attacks on fielded systems, with the notable exception of the work of Schuba et al. [SKK$^+$97] which studies a SYN flooding attack on TCP/IP.

2.5 Various miscellany

We now abstract various other works in the field of security that have had influence on our work in this dissertation, but that do not cleanly fit into any of the preceding categories.

Syverson [Syv97] calls for a relaxation in the worst case scenario when it comes to applying fault tolerant mechanisms within security[2]. He discusses the use of a probabilistic method of surviving an attack on a communication channel[3]. His argument then centres on viewing the correct processes and attackers as a pair of competing networks, and places the discussion within a game-theoretic model in order to try and calculate the processes' advantages in co-operating in groups against each other. While little is offered in the way of providing a realistic measurement of the applicability of such a model, we are not concerned with the implementation, but agree that models for computation other than worst case scenario can be sought to improve upon the application of fault tolerance within secure services.

One of the first papers to note the link between security and fault tolerance is by Turn and Habibi [TH86]. It sets out to deliver some minimal answers to questions regarding the compatibility of the two disciplines and the impact of the two design features on each other. Their first statement, that a security function is fault tolerant, if given the presence of a fault, the system's security policy remains intact is a useful notion. They ask if a security function can degrade gracefully, posing the question in terms of the DoD security evaluation criteria. They note that it should be possible for such graceful degradation to occur, if you allow for a downward migration from the system's evaluated division and class to function at a lower class or division. They note that in practice, this would require a design mechanism implemented with a high degree of modularity and diagnostics to test for the module's correct operation. They also make an interesting observation that protective redundancy lends itself better to security than corrective redundancy, although their use of the DoD criteria as a framework demonstrates their bias toward secrecy as the overriding factor behind this statement.

Gong [Gon93a] gives the arguments for and against the replication / dis-

[2] A similar argument is made by Meadows [Mea95, Mea94].

[3] This is an extension of the *two generals problem* [Gra78, pp. 465–472].

tribution of a secure authentication server. The main points made are, that in a non-replicated environment, the service can become a bottleneck, but in trying to design the system to avoid this, there needs to be careful consideration so as to not compromise the security of the system. He proposes an algorithm where the session key in a mutual authentication protocol is not generated by a single server or group of servers, but by the participants sending their candidate key through independent instances of the servers using a secret sharing scheme, and a concatenated hash of the two keys being used as the session key.

Chapter 3

Belief and Trust in Fault Tolerance

3.1 Overview

In this chapter we start by giving an overview of what we believe to be the tenets of fault tolerance as they apply to security protocols. We then view the differences that need to be taken into account when analysing belief in protocol specification using a modification of the BAN logic as a vehicle for our analysis. We conclude by looking briefly at a definition of trust used in the literature and how this changes in our environment.

3.2 Faults & Protocols

Within the body of fault tolerant literature [AL82, AL81, ALS79], they deal with a specific nomenclature when discussing the design principles in their examples. In this section we look at how these definitions translate when used in the context of security protocols. We use this as a stepping stone to get an intuition of how to challenge existing principles in the design of secure systems.

3.2.1 What is a fault?

When discussing a fault and its effect on the system we need to be clear about what we mean. By drawing up a modification of the standard terminology used in fault tolerance, we aim to shed some light on the principles that underline the mechanisms we propose in the remainder of this dissertation.

Our modifications are drawn upon the terminology proposed by Anderson et al. in various places [AL82, AL81, ALS79] which is used almost universally within the fault tolerant literature.

Fault, Error & Failure

We work backwards through the terminology chain defined in the literature. The final stage of error is referred to as a **Failure**. The definition as given in the literature [AL82] is given here:

Failure A failure of a system is said to occur when the behaviour of the system first deviates from that required by the specification of the system.

This definition relies on the notion of a specification for system behaviour. What exactly are we looking for in the specification of a security system? Taking a security protocol as an example, it has been noted in the literature [BAN89] that the function of a protocol is to allow its participants to achieve certain goals. In particular, we note that security protocols are generally designed with the goals of specific principals in mind.

Using a two-party authentication protocol [NS78] as an example, we note that A and B view the successful completion of a protocol as meeting certain goals upon completion of the protocol. Conversely S has no goals to fulfil by running the protocol. In this sense S is benign as far as the specification of the protocol is concerned. We introduce the term **Beneficiary** to describe A and B in relation to specification of the protocol, as they are the participants who aim to achieve the specified goals by running the protocol.

With this in mind we amend the definition to apply to a security protocol as follows:

Protocol Failure A failure of a security protocol is said to occur when the beneficiaries do not achieve the goals stated in the protocol specification.

In order for the system to fail there has to be an action prior to the failure, which the failure can be attributed to.

The system is defined to be encapsulated by a set of states which are changed by transitions. Both states and transitions are said to be either *valid* or *erroneous*. With the set of states represented by s_i with and transitions by t_i, then the state transition graph flows as shown in figure 3.1. An erroneous transition is one that moves the internal system state from a valid state to

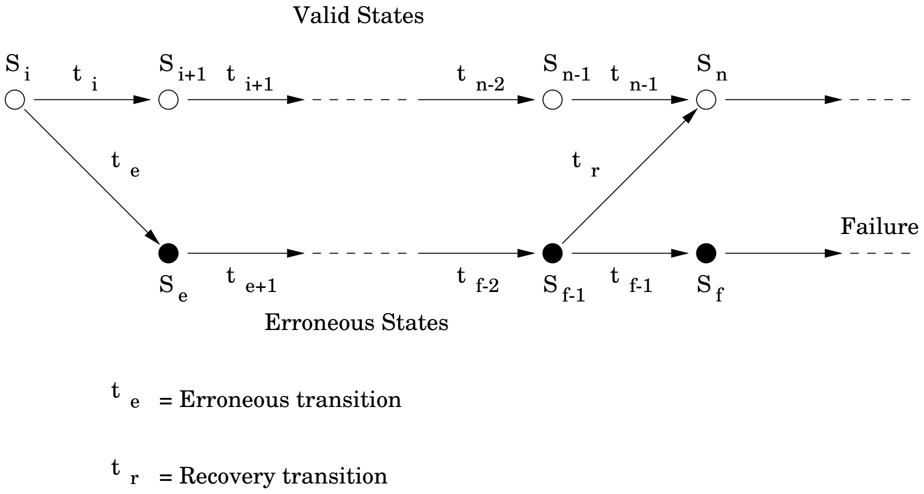

Figure 3.1: State transitions

an erroneous state. An invalid state is one which, if not transformed back to a valid state will result in the failure of the system.

Re-defining these in terms of a security protocol, we define the message content to encapsulate the **state** within the cryptographic protocol, and a **transition** is subsequently the movement at a participant from and incoming message to an outgoing message.

In the case of valid states and transitions, they are transitions which follow the protocol specification. For their erroneous counterparts we arrive at the following definitions:

Erroneous State A message, which could possibly allow a legitimate participant to make a transition, and, if not detected would allow a beneficiary to derive false goals.

Erroneous Transition A transformation from a valid message to an invalid message, taking the system into an erroneous state.

The terms used to describe the system prior to a failure are **Fault** and **Error**, and are defined are follows [AL82]:

Fault A fault is said to occur in the system, when an internal state is transferred from a legitimate state to an erroneous state.

Error An error is said to be when some internal system state is defective.

The four phases of fault tolerance [AL82] are identified as (i) *error detection*, (ii) *damage assessment*, (iii) *error recovery* and (iv) *fault treatment.*

We give a brief description of what each of these phases means in the context of a security protocol when taking into account our above definitions for states and transitions:

Error Detection Any participant in the protocol can perform error detection, but it is the beneficiary who is most likely to carry out this task. As is it their security policy which is violated, then they are likely – in some scenarios – to be the only participant in the protocol that notices the goals have not been satisfied.

Damage Assessment Damage assessment in the case of a security protocol is a short task. It is unlikely that much can be derived about what the "correct" state should be given the malicious nature of an attacker.

Error Recovery In the case of protocols this is likely to be a re-initiation of the protocol, with fresh state information (i.e., new nonces etc.)

Fault Treatment This is encapsulated by our definition of *Error recovery*, as there is likely to be little that can be derived about what the correct state of the system should be.

As noted in the section describing error detection, we believe that a beneficiary is likely to be in the best situation to detect an error. With regard to recovery and treatment, a beneficiary is also the principal who is primarily concerned with the knowledge that the protocol has been, or needs to be stopped or re-initiated. For this to be incorporated easily into protocols, we briefly describe a principle we call *Tightly Coupled Protocols* which allows for detection and recovery to be carried out by the beneficiary with minimum disruption to the protocol. Although we briefly describe it here, there are sections in the remainder of this dissertation which highlight our principle well, notably section 6.5.2. The basis of the principle involves the beneficiary acting as a 'hub' for the messages, with the beneficiary being involved in several rounds. This provides them with maximum exposure to the current state (i.e., message content). Although we do point out that this is not an ideal design methodology for some protocols, we do demonstrate in other parts of this dissertation scenarios where this can work effectively.

To clarify our discussion, we use some of the more well known protocols in the literature as examples:

Otway-Rees

Otway and Rees [OR87] proposed a simple authentication protocol. Using our definition above, the Otway-Rees protocol is tightly coupled around B. This gives B the maximum control over the execution of the protocol. This is counter-intuitive to the expectation in the majority of scenarios in which such a protocol will be initiated, where A – as the initiator – is likely to be the party most interested in successful completion of the protocol.

As both A and B are considered to be beneficiaries in this protocol, we introduce the notion of a *primary beneficiary* and *secondary beneficiary*[1]. In this case A can legitimately be regarded as the primary beneficiary, but is not the principal whom the tightly coupling is centred around.

What this demonstrates is that under this protocol, it makes it difficult for the primary beneficiary to carry out the constituent parts of fault tolerance (i.e., error detection etc.), because the protocol is not tightly coupled around that principal.

If we estimate the **inherent** fault tolerance in a protocol as a measure of the beneficiaries' ability to carry out the tasks necessary for fault tolerance, then we can conclude that this protocol does not contain a high degree of inherent fault tolerance.

Needham-Schroeder

In Needham-Schroeder [NS78], the tight coupling is centred around A, which is also the primary beneficiary in this example. This allows us to conclude that there is a greater degree of inherent fault tolerance in this protocol.

Wide-Mouth-Frog

The Wide-Moth-Frog [BAN89] is an example of a protocol that is not tightly coupled at all, where neither beneficiary has a high degree of control over the execution of the protocol.

[1] This can be expanded if more than two parties are regarded as beneficiaries.

3.3 Belief

In their BAN logic paper, Burrows et al. [BAN89] use a notion of *belief*[2] in order to analyse protocols. Their aim was to provide a means by which the goals of a protocol could be formulated and tested against a set of pre-conditions.

A strong assumption in their paper is that the principals involved in the protocol must trust the other principals involved in the protocol to behave correctly (e.g. that the Server in a Key Distribution Protocol does not mis-use the keys available to it). This assumption works well in an environment where the principals involved in a protocol trust each other to behave correctly. Our aim in this section is to look at the way in which these beliefs and goals have to change in an environment where assurance in the actions of others does not necessarily hold.

3.3.1 Belief and N-Model Redundancy

As a focal point for our study, we use the canonical example within Fault Tolerance of N-Model redundancy [AL81, pages 68–69]. This mechanism is a method of circumventing the failure of a component by replicating the component (N times) and then using a voter to provide a final result that masks the failure of a subset of the components.

When using this mechanism in security, its primary motivation is to allow the malicious failure of some subset of the components to be masked by the non-malicious components. In order to accommodate this into a logic such as the BAN logic, we need to make some changes to the structure of the logic. If we can understand how a principal's set of beliefs and actions upon that set are changed, then we can better understand the changes we need to make in the design methodology for such systems.

3.3.2 Key Distribution

To illustrate our example, we provide a simplified Public-Key Distribution Protocol and naively modify it to follow the N-Model redundancy principle mentioned above.

KDP protocol

$(1) A \rightarrow S : \{A, B\}_{K_S}$
$(2) S \rightarrow A : \{A, B, CERT_B, K_B\}_{K_S^{-1}}$

[2] We note that this is not meant to imply belief in the real world meaning, but is meant to dictate how a principal may act if a function were true.

Where, A and B are the principals, S is the public key distribution server, $CERT_B$ is S's certificate on B's public key, and K_S and K_S^{-1} are S's public and private keys respectively.

To modify this to a simple N-Model redundant service, let G be a group of servers, where S_i is an instance of the public key distribution server, where $i = 1 \ldots N$. Let $\Rightarrow$ denote a simple broadcast send and receive mechanism that A uses to contact the group of servers G.

N-Model KDP protocol

$(1) A \Rightarrow S_i : \{A, B\}_{K_{S_i}}$
$(2) S_i \Rightarrow A : \{A, B, CERT_B, K_B\}_{K_{S_i}^{-1}}$

A then compares the results from each of the servers S_i and if there is a clear majority that agree on B's key, then A uses that key to contact B.

3.3.3 Changes to the logical statements

As noted earlier, the postulates on which the BAN logic derives its goals are built around a belief in the actions of principals. We now look at how those beliefs change in the new environment.

The following postulates need no change in our revised model: $P \models X$ (*P believes X*), $P \triangleleft X$ (*P sees X*), $\#X$ (*X is fresh*), $\{X\}_K$ (*X encrypted under key K*), $< X >_Y$ (*X combined with formula Y*).

The other logical postulates need some change in their definitions, and we discuss each in turn below:

$P \mid\sim X$ *P once said X* - This postulate needs re-working slightly, which is necessitated by the notion of different types of principals we explore below, and how to attribute 'saying' in our new environment.

$P \mid\Rightarrow X$ *P has jurisdiction over X* - This postulate also rests on the assumption that a principal allowed to utter X during the protocol will do so correctly. Again, this does not necessarily hold for our revised model, and we will introduce new methods of achieving this later.

$P \overset{k}{\longleftrightarrow} Q$ *P and Q may use the shared key K* - In N-Model redundancy, we need to differentiate between a server that is replicated and a member of the group that forms the server. This means that we need to change the assumptions by which a shared key can be used.

$\overset{k}{\mapsto} P$ *P has a public key K* - We change this postulate to accommodate our re-definition of principals explored below [3].

$P \overset{x}{\rightleftharpoons} Q$ *X is a secret known only to P and Q* - This postulate clearly does not stand between an individual principal and a set of principals.

To allow the original aim of the logic to be achieved, we need to accommodate the points made above into the structure of the logic. We now expand on the structure of the logic, modifying the notion of a principal involved in a protocol and modifying the derivation rules to accommodate these changes.

3.3.4 Re-defining principals

In the original BAN logic, all principals are considered trustworthy, and the beliefs they derive are evolved from what are deemed to be the actions of the trustworthy party. This assumption does not hold if we are to consider faulty principals. We now re-define the types of principals in the model. Introducing two new principals of group and group member:

P : Principal : P is a normal principal as defined in the BAN logic.

G : Group : A group is a collection of principals, which have a recognised form to other members of the model. The abstraction of a group allows us to derive beliefs about the action of a replicated service that are otherwise unattainable if we independently assess the statements of individual members.

S_i : Group member : An identifiable member of a given group G. As noted in the discussion above, their ability to speak with authority in the terms of the logic is restricted, but their inclusion is necessary in order to form beliefs about the actions of the group as a whole.

By analysing pairwise communications between each of the principal types defined above, we define which of the logical postulates hold in each case. The logical postulates we re-define are those that relate to the use of keys and shared secrets, as these are the postulates from which we are able to derive the other postulates (e.g. P *says* X). Table 3.1 gives the relationships (A period is used to denote that the method of communication is not supported in our model).

[3] Although there are mechanisms for sharing a secret key between a group of processes in order to generate a shared signature, we do not explore the effect of this on our logic here.

		Receiver		
		P	G	S_j
	P	$P \xleftrightarrow{k} P$ $\overset{k}{\mapsto} P$ $P \overset{x}{\rightleftharpoons} P$	. $\overset{k}{\mapsto} P$.	$P \xleftrightarrow{k} S_j$ $\overset{k}{\mapsto} P$ $P \overset{x}{\rightleftharpoons} S_j$
Sender	G	. . .	. . .	. . .
	S_i	$S_i \xleftrightarrow{k} P$ $\overset{k}{\mapsto} S_i$ $S_i \overset{x}{\rightleftharpoons} P$	. $\overset{k}{\mapsto} S_i$.	$S_i \xleftrightarrow{k} S_j$ $\overset{k}{\mapsto} S_i$ $S_i \overset{x}{\rightleftharpoons} S_j$

Table 3.1: Pairwise sender / receiver – key / secret belief postulates

The central row demonstrates that we are unable to associate anything directly with a group G. To achieve these goals, we need to build beliefs about what a group says, without a group being able to directly say anything. To achieve this, we now form an extension to the rules that allow us to derive belief.

3.3.5 Threshold operators

We now include two new operators into the logic to capture the nature of group agreement protocols. These operators can then be used within the logic with the same assumptions that current operators (e.g., encryption) can be used, without needing to draw assumptions regarding concrete implementation into the logic.

The two operators relate to the agreement of the output of the group and to a threshold of response:

II - Signifies that a threshold of agreement between group members has been reached

$\trianglelefteq$ - Signifies that the threshold for response has been reached

The second operator signifies that there is a sufficient number of responses such that agreement might have been reached (i.e., there may be dissenting members in the group), and the first operator indicates that there are enough that agree on the same value in order that the value can be attributed to the group [4].

3.3.6 New derivation rules

Here is a simple example of how the new principals can be used in existing derivation rules:

$$\frac{P \models \overset{k}{\mapsto} S_i, P \triangleleft \{X\}_K^{-1}}{P \models S_i \mid\!\sim X}$$

But as we do not allow S_i to have jurisdiction, we cannot move forward to achieve $P \models X$. In order to achieve this, we have to define a new rule. The following rule is a modification of the existing jurisdiction rule, and makes an assumption that all S_i which once said something are all believed by P to part of the same group G.

$$\frac{P \models G \mid\!\Rightarrow X, P \models \amalg S_i \mid\!\sim X, P \models S_i \in G}{P \models X}$$

That is, if P believes that group G has jurisdiction over X, and that a threshold limit of members of the group S_i agree on X, then P believes X.

We now demonstrate the threshold for response and how it affects P's beliefs with respect to what the group says.

$$\frac{P \models G \mid\!\Rightarrow X, P \models \trianglelefteq S_i, P \models \not\amalg S_i \mid\!\sim X, P \models S_i \in G}{P \not\models X}$$

That is, if P believes that G has jurisdiction over X, and that a threshold of S_i have responded, but that a threshold of S_i have not said X, then P will explicitly not believe X. This is stronger than P simply disregarding X.

In order that P may reach some conclusion about the response of members of G, then P needs to be convinced that $S_i \mid\!\sim X$ is current, in order to achieve this we add a freshness rule.

$$\frac{P \models \forall S_i \mid\!\sim (X_i, Y), P \models \# Y}{P \models \trianglelefteq S_i}$$

[4] The values used in these cases are going to particular to the voting protocols used.

That is, if P believes that for all S_i which said X_i, that they also say Y (which is fresh – e.g., a request identifier) then P believes that a threshold of S_i have responded. Each S_i is allowed to say a different X for the response threshold to hold. We now give a rule (which we term the *combination* rule), that is used to move from response threshold to agreement threshold.

$$\frac{P \mid\equiv \trianglelefteq S_i, P \mid\equiv \forall S_i | S_i \mid\sim X \ \forall S_i | S_i \in G}{P \mid\equiv \ \amalg S_i \mid\sim X}$$

That is, if P believes that if a threshold of S_i responded with the same X, then the group G said X. It is this rule that allows us to derive belief about what a group said.

message meaning

As stated earlier, we do not have an explicit means for a group to be able to say anything, this is because we do not allow a key to be associated with a group. We now expand on the message meaning rules for both private and public key to include derivation rules that include group members.

private keys

$$\frac{P \mid\equiv S_i \overset{k}{\longleftrightarrow} P, P \triangleleft \{X\}_K}{P \mid\equiv S_i \mid\sim X}$$

That is, if P believes that k is a good key between S_i and P, and P sees X encrypted under k, then P believes that S_i said X.

The following two rules carry on from this, allowing S_i to believe that P and S_j said X:

$$\frac{S_i \mid\equiv P \overset{k}{\longleftrightarrow} S_i, S_i \triangleleft \{X\}_K}{S_i \mid\equiv P \mid\sim X}$$

and:

$$\frac{S_i \mid\equiv S_j \overset{k}{\longleftrightarrow} S_i, S_i \triangleleft \{X\}_K}{S_i \mid\equiv S_j \mid\sim X}$$

public keys

$$\frac{P \mid\equiv \overset{k}{\mapsto} S_i, P \triangleleft \{X\}_{K^{-1}}}{P \mid\equiv S_i \mid\sim X}$$

That is, if P believes that k is a good public key for S_i and P sees X encrypted under k^{-1}, then P believes that S_i said X.

This rule carries to S_i, allowing S_i to believe that P and S_j said X:

$$\frac{S_i \mid\equiv \overset{k}{\mapsto} P, S_i \triangleleft \{X\}_{K^{-1}}}{S_i \mid\equiv P \mid\sim X}$$

and:

$$\frac{S_i \mid\equiv \overset{k}{\mapsto} S_j, S_i \triangleleft \{X\}_{K^{-1}}}{S_i \mid\equiv S_j \mid\sim X}$$

shared secrets

$$\frac{P \mid\equiv S_i \overset{Y}{\rightleftharpoons} P, P\triangleleft <X>_Y}{P \mid\equiv S_i \mid\sim X}$$

That is, if P believes that Y is a good shared secret for S_i and P, and P sees X combined with Y, then S_i said X.

This rule also carries to S_i, allowing S_i to believe that P and S_j said X:

$$\frac{S_i \mid\equiv P \overset{Y}{\rightleftharpoons} S_i, S_i\triangleleft <X>_Y}{S_i \mid\equiv P \mid\sim X}$$

and:

$$\frac{S_i \mid\equiv S_j \overset{Y}{\rightleftharpoons} S_i, S_i\triangleleft <X>_Y}{S_i \mid\equiv S_j \mid\sim X}$$

3.3.7 Belief and control

The exercise of adapting the BAN logic to accommodate N-Model redundancy was done in order to explore what happened to the belief of ordinary principals (e.g. clients of a server S in the cryptographic key distribution scenario) by adopting such a model.

We see from the changes we made to the logic, that the belief moves from the actions of the principals themselves to the threshold mechanism. It is the combination rule that allows a principal to move from merely observing that members of the group said X to believing what they say. This is crucial if we are to achieve goals such as $A \models A \overset{k}{\longleftrightarrow} B$.

By looking at the authentication protocols listed in the literature, and studying how they are able to achieve their stated goals, we observe that they are derived from the construction of messages in such a manner that a principal involved in a protocol run can use the evidence presented by the mere fact that he knows the origin of the message to make judgement about its correctness. Accepting the N-Model redundancy view presented above, we note that this no longer holds. Instead, the belief now relies on the principal's understanding of the component of trust afforded to the mechanism that combines the output (i.e., in a traditional N-Model scenario, the *voter*).

By taking this mechanism for controlling the principal's belief and placing it under the control of the principal, this gives the principal greater confidence in the result achieved.

3.4 Trust

In their paper on trust, Christianson and Harbison [CH96] discuss several things which trust is not. One of the points of their argument is that "**Trust is not Reliance**". In this they argue that just because A relies on a principal to carry out some action, this does not imply that A trusts that principal. We agree with their conclusion.

The inclusion of techniques derived from fault tolerance in security protocols is there primarily as a goal of reducing this notion of *A trusts B* (or *A relies upon B*) even further.

Given that A now does not even rely upon B to complete their actions properly, where does this leave us?

As we see from our logic, A needs to have confidence in the mechanism for indirectly determining confidence in the fact X (because B says so), as she does not have confidence in X directly from the fact that B says X. In this sense A is moving her trust from B into the mechanism for scrutinising B's actions.

Given that A now trusts a mechanism in which we can detach from B, the next logical step is to take the mechanism under A's control directly, for surely, if A is to trust anyone, then it must be herself.

Reverting to our use of N-Model redundancy as an example, figure 3.2 shows the difference we have between reliance in a standard cryptographic protocol, and one in which fault tolerance is employed.

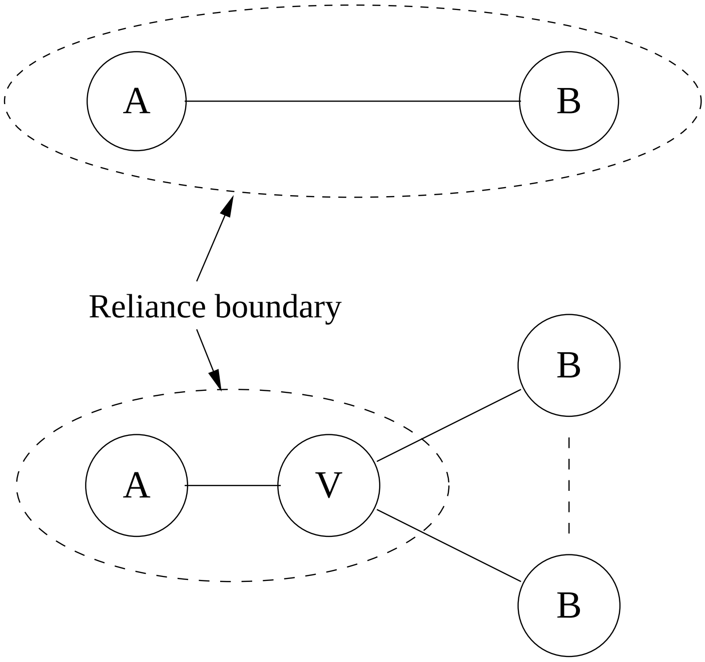

Figure 3.2: Reliance in cryptographic protocols

Dobson and Randell [DR86] criticise the Trusted Computing Base (TCB) view of secure computing, because its model does not always describe the user-perceived model of the security-critical behaviour of the system. They also note that a TCB view is at odds with the concept of a distributed environment. If we regard the canonical key distribution mechanisms studied in the literature, we note that they all follow this example of having a TCB.

By allowing the client in a distributed environment to control the security interface in a manner which fits their security policy, we can move to a system where there is less reliance on the TCB.

The notion of reducing reliance in single components is not new to security. What we believe to have illustrated with our example in figure 3.2, is that the way in which a client can satisfy themselves that their security policy is being complied with, can exclusively be handled by the client in some systems.

Such mechanisms are not achievable in every scenario. There are instances where it may be impossible to move to a system where designing the system to not have a single point of failure is impossible. Nevertheless, it is an interesting proposition that we might be able to design secure systems that "farm-out" the ability to dictate the course of events on which the client relies.

We conclude by noting that replication is currently used not to reduce reliances, but to increase the assurances given about the existing reliances.

Chapter 4

Choice and control in protocols

4.1 Overview

In this chapter, we look at ways in which the control of the protocol can be used to offer advantages in terms of security. We start off by looking at the well trodden path of timeliness in authentication, offering a new perspective on the relative merits of timestamps and nonces. Secondly, we offer a view on how anonymity can be improved by having those that wish to communicate with each other control the structure of the communication mechanism, and we conclude the chapter by looking at using anonymity as a means of defending against a new class of denial of service attack.

4.2 Nonces v. Timestamps

It has long been a topic of discussion whether to use time-stamps or nonces as part of an authentication protocol. The relative merits have seen the light of day in many a publication [NS78, DS81, NS87], which highlights the importance of choice in security protocols.

In their work on timestamps, Denning and Sacco [DS81] discuss the use of timestamps in protocols. We propose another attack that re-enforces the subtlety of the balance in this area.

If we consider an authentication mechanism similar to that used in Kerberos [NT94], there is a subtle management of trust in this scenario, where the server is dependent on the authentication / authorisation mechanism in more ways than one. We consider a novel yet enlightening attack that allows the authorisation mechanism to hand off responsibility for its duties without compromising itself to the principal with which it colludes.

Many of these subtleties are only found in the protocols if we vary the reliance placed in the players, which is rarely done in the literature with regard to authentication and authorisation.

Consider a scenario where server S is expecting the authentication server AS to handle all ticket requests, which S will then act upon in good faith. If user A approaches AS for ticket, then S has to be sure that AS is doing its duty correctly.

If we de-couple the trust that S places in AS, then we call attention to a new attack which we coin the **Lazy Warden Attack**. Using a message structure similar to that used by Kerberos, the normal protocol proceeds as follows:

$(1) A \rightarrow AS : A, S$
$(2) AS \rightarrow A : \{A, S, K_{A,S}, \{A, S, T, K_{A,S}\}_{K_{S,AS}}\}_{K_{A,AS}}$
$(3) A \rightarrow S : \{A, S, T, K_{A,S}\}_{K_{S,AS}}, \{Y, A\}_{K_{A,S}}$
$(4) S \rightarrow A : \{X, A\}_{K_{A,S}}$

Where Y and X are service defined parameters, and T is a timestamp.

In this case S does not want to have to manage all the authentication mechanism itself, and is willing to rely on AS to provide some degree of service, but does not want to rely on AS more than is required.

We now demonstrate a simple attack which AS can be a conspirator to, while limiting the amount of control he is relinquishing.

AS is willing to provide the authentication mechanism, but does not want to pay for the facility of providing a full on-line service, the attack proceeds as follows:

AS comes to an agreement with a conspirator C to provide the on-line portion of the service, while AS retains control of the generation of tickets. We assume a system with a similar property to Kerberos, where the validity window of the timestamp is quite large (typically 5 mins).

At the start of each period (e.g., once a day) AS generates a set of valid tuples $< A, S, T, \{ticket\} >$ (where $\{ticket\}$ contains the information sent in the second message of the above protocol). Each of these is encrypted by $K_{A,AS}$, which is not known to C. There are enough of these to allow C to reply to a request on AS's behalf without the other participants of the protocol noticing that AS is not performing its obliged duty.

With memory being plentiful, it would take very little for C to store all the tuples required to fake AS's presence on the network.

The important point we note about this attack, is that it is possible to carry it out without AS being able to entrust any $K_{A,AS}$ to C. This means C cannot by himself generate any new valid tokens. Also, the tokens already in C's possession are only of use to the users who they have been validly formed for, and C gains nothing from having them in his possession. It also means C never gets to see any of the $K_{A,S}$ session keys and thus cannot eavesdrop on subsequent communications between the client and the service.

We now present two alternatives to this protocol, reverting to a nonce based mechanism. The first causes S to perform more communication steps than the second, while still only requiring 4 messages. While the second increases the message count to five, but has a number of different properties. In both AS only deals with the same number of messages.

4.2.1 Variant One

In the first variant of our protocol to defeat this attack [1], we need to add the following information:

N_S - A nonce generated by S for each request for service.

G/R - A flag set by AS to denote if the authentication request has been granted or denied.

The protocol itself is as follows:

$(1) A \rightarrow S : A, S, Y$
$(2) S \rightarrow AS : \{A, S, N_S\}_{K_{S,AS}}$
$(3) AS \rightarrow S : \{A, S, N_S, G/R, K_{A,S}\}_{K_{S,AS}}, \{A, S, K_{A,S}\}_{K_{A,AS}}$
$(4) S \rightarrow A : \{A, S, K_{A,S}\}_{K_{A,AS}}, \{A, S, X\}_{K_{A,S}}$

This protocol achieves the same goals as the first one, but has the drawback that Y is sent in the clear. This might be unacceptable in some situations. The problem arrives from the fact that A and S do not yet share a session key. In order to solve this problem, we develop a second variant to the protocol.

[1] The objective here is to ensure that AS is really live.

4.2.2 Variant Two

The second variant uses a cyclic protocol structure, with Y being sent to S in the same message as the session key, allowing Y to be encrypted for secrecy.

$(1) A \rightarrow S : A, S$
$(2) S \rightarrow AS : \{A, S, N_S\}_{K_{S,AS}}$
$(3) AS \rightarrow A : \{A, S, N_S, K_{A,S}\}_{K_{S,AS}}, \{A, S, K_{A,AS}\}_{K_{A,AS}}$
$(4) A \rightarrow S : \{A, S, N_S, K_{A,S}\}_{K_{S,AS}}, \{A, S, Y\}_{K_{A,S}}$
$(5) S \rightarrow A : \{A, S, X\}_{K_{A,S}}$

The G/R flag does not need to be used because of the implicit declaration that the request has been granted when S receives a message with a fresh nonce and key.

While this protocol increases the message count, the number of messages the service needs to process stays the same as the previous protocol, but has one advantage to the service. If A is refused authentication from AS, then S only has to deal with two messages, while in the first variant, S has to be party to three messages.

4.2.3 Discussion

The main disadvantage of this scheme is that the service has to be party to all requests whether valid or invalid. This does not compare well with the timestamp based protocol, where invalid requests are never brought to the attention of S.

Our purpose in highlighting this new attack was:

- That the control offered by nonces in protocols can be used as a factor in protocols to reduce the reliance that one participant in the protocol has upon other members that take part in the protocol. In the example we provide, the assumption is that S is willing to rely upon AS to distribute tokens to correct members, but does not want to rely upon AS in manner where AS can break a *duty-of-care* (i.e., AS can carry out the above protocol with the aid of the a colluding third party unbeknown to S, without AS compromising himself) which we believe to be an interesting point in itself.

4.3 Group anonymity

We present a protocol that allows members of a group to communicate without external traffic analysis being able to discern which members are communicating at any particular time. Our method is extremely lightweight and has the advantage over existing MIX-net remailers in that the protocol is invulnerable to analysis under duress, even to a complete compromise of all other parties communicating in the system.

4.3.1 Introduction

We introduce a simple protocol that allows a community of users to communicate with each other that is resistant to traffic analysis. The primary goal is to stop an external source form discovering which members of the group are communicating pairwise, at any particular point in time. Our method is resistant to external duress being applied in pursuit of this information. It is of no significance to us that the external observer already knows the group membership.

As a context to our discussion we examine the scenario where a group of mobsters wish to communicate without the F.B.I. gleaning any details regarding the information flow between the members.

The F.B.I. will already be aware of the internal membership of the group, and the mob will know they are being watched. They could set up a dedicated communication network, but this would be expensive, and provide an easy point at which to launch a denial of service attack. We propose a method whereby they use nothing other than publicly available communication media.

There already exist different methods of supplying sender and receiver anonymity. Chaum first proposed methods for providing anonymity through the use of MIX-nets [Cha81], a mechanism already used in providing anonymous re-mailers. Although publicly available, we do not use them here for one reason – the F.B.I. could obtain a court order forcing the owners of the remailers to hand over their logs to the authorities. In these scenarios the remailers are liable to be subject to duress by legal enforcement, and subsequently the mobsters would be unable to rely upon the integrity of the remailers.

A similar mechanism discussed in the literature that could fall in the face of duress is Onion Routing by Syverson et al. [SGR97].

The second mechanism for providing anonymous communication put forward by Chaum is that used to solve the dining cryptographers problem [Cha88]. This offers an information-theoretically secure way of solving the problem, but relies on the existence of a reliable broadcast network for the participants to use. This is an unrealistic goal, and would force the mobsters to use mechanisms other than those at hand. This idea has been expanded upon by Waidner [Wai89] so that the network need not be reliable, but at most fail-stop. This unfortunately has a high degree of complexity in its implementation.

The method we use is in line with the idea described by Pfitzmann and Waidner as RINGs [PW87]. It uses a similar mechanism to that used in Token Ring networks, where the ability to write is passed around the ring in a round robin fashion.

4.3.2 Basic structure

The communication ring is set up such that it is a cyclic graph containing all group members. We assume that they can meet confidentially, but that this itself is a difficult task to organise, and precludes the ability to meet regularly enough to make it the choice method of communication.

At an initial meeting they all exchange their public keys (formed before hand in private – in our example we will assume the use of R.S.A. public-key encryption [RSA78]), and they organise the communication path by some suitable means – such as drawing straws.

Before we describe the protocol itself we will introduce the notation:

B	:	a group of n mob members
B_i	:	the i -th member, where $i = 1 \ldots n$
K_i & K_i^{-1}	:	B_i's public and private keys respectively
M_i	:	a message destined for the i -th member of the group
$\{M_i\}_{K_i}$	:	message encrypted under the public key of B_i

In practice, the encryption will be carried out with a shared key algorithm and the secret key encrypted using the public key. We use the $\{M_i\}_{K_i}$ abbreviation in order to save space in our description.

The protocol is implemented by passing a token round the ring. This token implements the right to send on the channel, and the content of the message stream in one. The token is received at B_i, processed and then forwarded to member B_{i+1}.

We outline the content of the token below, where P_i is used to denote padding added by B_i:

$$\{\{M_1\}_{K_1}, \{M_2\}_{K_2}, \ldots, \{M_i\}_{K_i}, P_{i-1}\}_{K_i}$$

When B_i receives the token he strips the outer layer of encryption and does a trial decryption over all messages. In the above example, he finds a message in the content for himself (M_i) – we assume that the natural redundancy in the plaintext will identify a message.

B_i replaces padding P_{i-1} with a new padding P_i, and the whole packet is then encrypted with B_{i+1}'s public key and sent on to him.

B_i does not remove M_i from the token. It is left for the sender to remove once the token arrives back with him. This is done in order to stop a subset of 4 members $(B_{i-1}, B_{i+1}, B_{s-1}, B_{s+1})$ from colluding in order to deduce that B_s has sent a message to B_i. If need be B_s can require a receipt of delivery from B_i by signing the message. B_i will then add a suitably encrypted receipt to the token, which B_i will be responsible for removing. The padding – chosen randomly for both length and content – is there as a safeguard against eavesdroppers comparing message lengths on consecutive links.

In this scenario, if the two members – B_{i-1} & B_{i+1} – either side of the sender collude, they can compare the unencrypted contents of the token, to see if there are any new messages. As a precaution against the F.B.I. having been able to infiltrate the two mob members either side of B_i, he adds random messages to the token, again removing the message on its return. The frequency of such messages inserted into the token will be a function of the amount of genuine traffic sent by B_i, with the number of messages generated by a particular mobster over the course of a day falling within a pre-determined distribution.

4.3.3 Additional details

We will now expand upon the functionality of the system, and give a simple description of a how it could work in the real world.

One problem with the protocol as it stands is that if one group member wishes to communicate the same message to all other group members, then it will involve a large number of additional messages and computational overhead. In order to avoid this problem, we have the group generate a group key K_g, and each group member will retain a copy of this key. The key would then be used to do a second set of trial decryptions over the messages in the token. To provide a speedup, a conventional secret key algorithm could be used for group messages.

Another addition to the protocol we make is that the token should have a fixed processing time t at each group member. This time should be selected according to the expected number of trial decryptions, and also allow for the time to modify the token (adding new messages and padding etc.). This will stop an eavesdropper from being able to deduce any useful information according to the duration the token was present at any given group member.

In the case where B_{i+1} is unavailable, then B_i will send the message on to B_{i+2}, with a group message in the token explaining B_{i+1} unavailability. If B_i is deliberately trying to leave B_{i+1} out of the loop, then B_{i+1}will know quite soon, due to the bounded circulation time of the token.

What makes this system so simple is its ability to be implemented on top of existing mechanisms. An example would be to add this functionality to an existing e-mail system. The majority of e-mail systems have a forwarding function built in to them, this could be programmed to interpret the token according to our given specification. E-mail encryption products are also readily available, and could be linked to the forwarding software.

The protocol could be set up to run automatically, with the e-mail software storing all incoming messages in a folder for the mobster to read at his leisure. The software would check a separate folder for any messages to be added to the token upon its arrival.

An extension that would allow us to keep the number of messages present in the token bounded, would be to form a queue of outgoing messages in the folder, with the software picking the head of the queue each time, thus keeping the processing time to within a known envelope. If the queue was empty, then the software could insert a garbage message. This would increase this scheme's resilience to traffic analysis in the face of internal members colluding.

4.3.4 Conclusions

In this section we have detailed an extremely simple protocol that provides anonymity to a group of users. We outline a scenario where the protocol could be of particular use, notably where a group of users wish to communicate in a manner resistant to traffic analysis from external eavesdroppers, but where the existing MIX-net remailers might not be able to provide the resilience needed under duress.

The protocol only requires the participants to rely on those with whom they wish to communicate. This reliance is limited to relying upon the other group members to forward the token without disturbing messages that are not intended for them.

An advantage of this mechanism it that is can be set up to run over existing e-mail software, and provides a continuous messaging service within a tn-window (where t is the pre-determined duration which a token must be stationary at a group member, and n is the number of group members).

4.4 Choice and Denial of Service

In this section we view the influence of choice and anonymity on preventing a denial of service attack. Although noted in the literature as a concern within security systems, little has been written on the subject of denial of service, with the notable exception of work done by Needham [Nee94] and Gligor [Gli86]. Currently, the major concerns within Denial of Service are maintaining availability of the server (especially Needham's work concerning the vault) and maintenance of fair play between users of a service. Little is said with regard to the problem of the server itself launching a denial of service attack.

It is an interesting variant of this problem that we introduce and attempt to solve.

Consider a system providing a service to a set of users. We are concerned about a variant of denial of service, where the service can selectively deny service to some portion of the users with a high degree of accuracy. We coin the phrase **Selective Denial of Service** to describe this scenario.

More formally: A system is resilient to a **Selective Denial of Service** attack if it consists of a service S (potentially a set of services $\mathcal{S}$), and a set of users $\mathcal{U}$, where the service should not be able to selectively deny service to any particular member A within the set of users $\mathcal{U}$ with greater probability than random choice.

We do note two exception to the above definition:

- It its possible for S to deny service to A by denying service to all members of $\mathcal{U}$ simultaneously, by closing down completely, but this goes outside the bounds of selectivity used in our definition.
- It is also possible for S to deny service to A if it can manipulate the group membership properties of $\mathcal{U}$, such that A is the only member of the group. Again, we consider this threat scenario beyond the scope of our work.

4.4.1 Building a solution through anonymity

Our solution to this problem draws upon the use of anonymity mechanism for use in different requirements. In his work on digital cash, Chaum [Cha82] proposed a mechanism that allowed a user to generate electronic cash in a manner that preserved the user's anonymity during spending.

The building block of the system is that of *blind signatures*, where the issuing bank is asked to sign one of a set of messages that have been converted into unrecognisable values by the use of a *blinding factor*, where the factor is chosen by the user. The issuer of the digital money is allowed to ask the user to un-blind a portion of the requests to ensure that they are not about to sign a message allowing the user to spend more electronic money than they should.

We take this mechanism presented by Chaum and modify it to provide a solution to our problem.

Below we provide two solutions to the problem. Our first attempt, although a more naive variant, has one interesting property which is not present in the second. We developed the second after realising some of the shortcomings of the first, to provide a more complete and robust solution.

4.4.2 A first cut

Before describing our protocol we first of all define the players involved in our scenario:

Z The Authorisation server : Provides the users with the tokens that allow them to use a service S.

N The Authentication server : Checks the authenticity of the users with regard to their membership of the system.

S Server : Any of a set of servers $\mathcal{S}$.

A User : A uniquely identifiable member of the set of legitimate users $\mathcal{U}$.

G Group : A non-empty, identifiable subset of the legitimate users $\mathcal{U}$, to which A is a member.

In both our protocols, A wishes to receive some service from a provider S. The messages that make up the first instance of our protocol are shown in figure 4.1. The protocol is organised into three rounds. In each round the user A communicates with Z, N and S in turn. The messages exchanged in each protocol round and their significance are described below:

Round 1 : A to Z

This part of the protocol is used to allow A to request a ticket to use a service S from the Authorisation mechanism for that service.

A starts off by generating a set of messages m_i (where $i = 1 \ldots j$) of the form "I am a member of G and wish to access service S". A then generates $2j$ random numbers and splits them into two sets U_i and V_i, using the product of the random numbers to generate the following:

$$A : t_i = m{K_i}^e \ mod \ n \ ; \ K_i = U_i * V_i$$

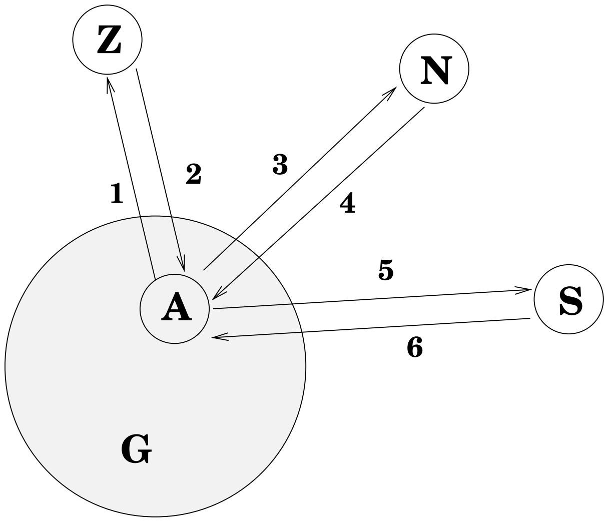

Figure 4.1: The first protocol to counter Selective Denial of Service

Where n and e are components of Z's public key[2].

A then sends to Z the set of t_i's and receives one of these raised to d (the secret component of Z's public key). For brevity we drop the section of the protocol where Z requests for $j-1$ of the messages to be un-blinded.

$(1) A \rightarrow Z :< t_1, \dots, t_j >$
$(2) Z \rightarrow A : t_r^d \ mod \ n$

Round 2 : A **to** N

In this section of the protocol, A seeks to bind her authentication as a member of a particular group G to the authorisation she has to use service S.

A takes the message received from Z in the previous round and partially un-blinds the message using one of the two random numbers used to blind it to generate a new message m'.

$$A : m' = t_r^d / U_r \ mod \ n$$

A then presents m' to N along with her credentials that she is A and that she is a valid member of G. A's details are checked by N, and if satisfied, N hands back a signed token that confirms A's identity and membership of G. Note that, even if Z and N collude at this point, they cannot correlate A's

[2] As in the original work by Chaum we assume R.S.A. [RSA78] as the public key cryptosystem.

m' to the original t_r, because of the use of the two-phase random blinding, this makes it impossible for them to selectively deny A access to S.

$(3) A \rightarrow N : A, G, m'$
$(4) N \rightarrow A : \{A, G, m'\}_{K_N^{-1}}$

Round 3 : A to S

In this section, A requests the service from S, providing S with all the necessary information that both previous stages of the protocol have been completed successfully, along with evidence that A is a member of G and that as a member of G she is able to use S.

A does this by sending the token received from N at the end of the previous round, and also revealing the second of the random numbers used to blind the request. Q is any request dependent material, and T is the response to Q. Although in some services message 6 will not be necessary.

$(5) A \rightarrow S : \{\{A, G, m'\}_{K_N^{-1}}, V_r, Q\}_{K_S}$
$(6) S \rightarrow A : \{T\}_{K_A}$

The main drawback with this protocol, is that the server itself can still mount a selective denial of service attack against A. This is because, when we were first contemplated this attack, our main concern was the authentication and authorisation mechanism. One of the reasons for attempting a second protocol was to negate the threat at all points in the system.

The second point that makes this protocol less desirable, is that the authorisation is carried out before the authentication. This allows a corrupt user of the system to potentially gather information about what groups were able to use which services without actually being a member of the group. There is also the problem that, by having authorisation before authentication, a user who is not even a member of the system can slow down the authentication process for legitimate users[3].

4.4.3 A second cut

In the second instance of our protocol, A's interaction with the authorisation and authentication services are reversed. We also highlight the fact that A is potentially a member of a number of various groups, and indicate this through explicitly adding a second group G' into the description.

[3] We thank Bruno Crispo for highlighting the presence of this potential cause for harm.

The new protocol design is shown in figure 4.2, the players are the same, with the addition of the second group. The protocol proceeds with the same three round structure as mentioned before, and we describe each of the rounds below:

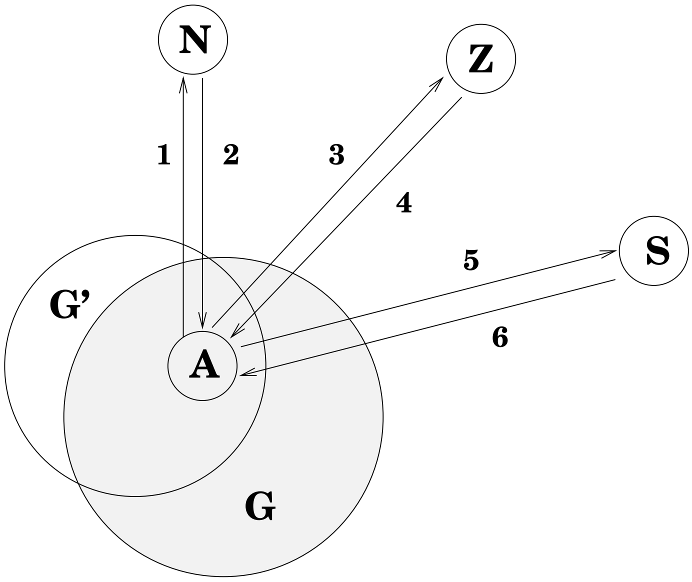

Figure 4.2: The second protocol to counter Selective Denial of Service

Round 1 : A **to** N

The reason we highlight the difference between G and G' in this section is that A needs to carry out each section of this protocol separately for each group of which A is a member and for which she wishes to use the associated "right" incorporated within the group at any given time period.

Thus $\forall G : A \in G$, A generates m_i (where $i = 1 \dots j$) of the form "As a member of G my pseudonym is 'Fred' ". A then generates a set of j random numbers (K_i), and then computes:

$$A : t_i = m_i {K_i}^e \ mod \ n$$

The set of all t_i is then sent to the authentication service, with the singing and un-blinding being carried out as before. Again, for brevity we omit the part of the protocol which reveals the blinding.

$(1) A \rightarrow N : A, < t_1, \dots, t_j >$
$(2) N \rightarrow A : t_r^d \ mod \ n$

The main difference between this and the first protocol in this section is the explicit use of a pseudonym[4].

Round 2: A **to** Z

To reveal which pseudonym A has been left with, she then performs the calculation:

$$A : m_r = t_r^d / K_r \ mod\ n$$

A then sends this signed pseudonym, along with the name of the service S she wishes to use to the authorisation server. Z then checks that a member of G is able to use S and generates a valid token for A by signing the request and sending it back.

Again, even if Z and N collude at this point they cannot recover any useful information because of the blinding mechanism.

$(3) A \rightarrow Z : S, m_r$
$(4) Z \rightarrow A : \{S, m_r\}_{K_Z^{-1}}$

Round 3: A **to** S

To use the service, A sends the valid token to S, along with the request information Q as before. Note that, to secure the reply with T, A has to include a session key for S to use in order to encrypt message 6, because A's own public key cannot be identified by S at this point.

$(5) A \rightarrow S : \{\{S, m_r\}_{K_Z^{-1}}, k, Q\}_{K_S}$
$(6) S \rightarrow A : \{T\}_k$

4.4.4 Discussion

We now discuss the relative merits of the two protocols described above.

As pointed out earlier, the first protocol has two problems associated with it, namely that the authorisation is done before the authentication and that the user has to rely on the service S itself not to carry out a selective denial of service attack.

It does however have one distinct advantage over the second protocol. If the system requires some form of *per-user* charging with regard to the use of resources, then it is far easier to do with the first protocol. S can simply

[4] Although we use a name here for illustrative purposes, a 128-bit random number could be used in practice.

keep information on which members of $\mathcal{U}$ have used its services, which can then be passed on to the charging mechanism.

Achieving the same with the second protocol becomes more complicated, with some form of *post-use* un-blinding having to be done by the charging mechanism, where S stores the pseudonyms instead of the actual names. One way to enforce this would be to make authorisation of today's pseudonym reliant on the un-blinding of yesterday's pseudonym.

Under the second protocol, in order to try and remove the threat of accumulating usage statistics on each allocated pseudonym, which could be used to generate patterns that can be used to try and detect user behaviour – these patterns then being used to try and spot users in future runs – we could have the first round only reveal a subset h (where $1 < h < j$) of m_i's with the remaining $j - h$ being signed as valid pseudonyms (although this will add overhead to the first round, as this is effectively a single sign-on facility, it would only need to be carried out once every sign-on period, e.g., once a day).

4.5 Conclusions

Our aim in this chapter was to continue exploring the approaches by which control over components within the protocol can act as a means to achieve fault tolerance within a secure environment.

We started off by re-visiting the classic argument regarding the use of timestamps v. nonces in authentication protocols. We highlight the fragile nature of the trust relationships with a simple yet novel attack. We turn the use of the nonce on its head. Originally used to assure the recipient of the freshness of the message in a protocol with another trusted principal, we use it here as a means to remove the need for one participant in the protocol to wholly trust the other participant with regard to carrying out part of its duty. What this shows is that by allowing a participant to control part of the execution of the protocol, they can narrow the trust boundaries they place with regard to other members of the system.

In our second analysis, we demonstrate the extent to which control of the execution of a protocol can effect the guarantees provided by the protocol. If a set of individuals wish to communicate anonymously, then there already exist technologies that can deliver this with a reasonably high degree of robustness. We use our notion of protocol control to go one step further, providing water-tight guarantees even in light of the corruption of all other participants in the protocol, which – in existing proposals – would render the legitimate players open to attack. The protocol we propose is lightweight and efficient in terms of its execution.

We finish with an example which we consider to be a new variant of denial of service attack. We place the client in a position to reduce the ability of the service to act maliciously. By using a modified version of an existing anonymity scheme, we demonstrate that by controlling the information flow in part of the authentication mechanism, the client is able to satisfy himself that the service cannot single him out in an attempt to deny him service. At the same time, the authentication mechanism is still able to fully carry out its duty.

Chapter 5

Adaption of Models

5.1 Overview

In this chapter we view different models and conventions used within fault tolerance, and their adaption to security applications. We look at how some of these models can be used to help security, and how some of the concepts need changing in their transportation between environments.

5.2 Introduction

We look at 3 different conventions described in the fault tolerant literature and see how their adaption into a security environment can add functionality to security. We also notice, that while some conventions translate into their new environment with ease, others require some changes to be made.

5.3 Call-back in revocation systems

Revocation in a distributed environment can be regarded as either being *on-line* or *off-line*. On-line systems follow the practice of allowing a client to query the revocation server directly to achieve a high degree of timeliness with regard the assurance of the revocation. This is difficult to achieve in practice, and in some circumstances can lead to the Certification Authority (C.A.) being more susceptible to an attack. For this reason, most proposed systems allow for an off-line mechanism, which allows the C.A. to generate the revocation material, and then they periodically update the information by sending it to a *directory*. These directories do not need to be trusted as much as the C.A. and can also be replicated to provide high availability. In off-line systems, there is a balance to be struck, between having a longer refresh period (thus allowing for cheaper update mechanisms), and a shorter refresh period (providing a more timely service). We propose a mechanism that can be used to augment any existing revocation service, which provides the best of both worlds.

A well known mechanism in fault tolerance is to signal a process of an undesirable external event in another part of the system, which is commonly termed as an *exception* in the literature [AL81, pages 77–88]. This process can then take any necessary corrective action internally.

Our proposal is based on the notion that by overlaying such signalling mechanisms, we can have the same degree of granularity as an on-line system using the structure of an off-line system. We note that these mechanisms are only of real relevance for use in systems where the requests sanctioned by servers on behalf of clients in the system are reversible.

5.3.1 Lists, Trees & Systems

We highlight some of the current literature on revocation mechanisms and then show how our adaption can be incorporated into any of these. Three current proposals are: work done by N.I.S.T on Certificate Revocation Lists (CRL) [U.S95], work done by Micali on Certificate Revocation Systems (CRS) [Mic96] and work on Certificate Revocation Trees (CRT), initially demonstrated by Kocher [Koc] and expanded upon by Naor and Nissim [NN98].

The standard players, commonly defined in the literature are:

- **Certificate Authority (C.A.)** - The trusted party who generates the certificates, and is also responsible for issuing the revocation material.
- **Directory** - A relatively untrusted service, that is used as an on-line repository of revocation material. Can be replicated to provide high availability.
- **User** - The end user who queries the directory to find out the latest revocation information associated with a certificate.

In each of the mechanisms (CRL, CRS, and CRT), the CA is the party who issues the original certificates for a public key. It is also responsible for issuing the revocation material if the validity of the public key is called into question before the expiry period placed in the certificate itself.

This revocation information is then periodically disseminated to the replicated directory. These provide the on-line part of the service without the user having to trust them to do anything more than disseminate the information.

When the user requires to check the validity of a public key, they then poll one of the replicated directories to retrieve the most up to date information. The user will then receive, either a proof of revocation, or a proof of an absence of revocation (depending on which revocation system is used).

The inherent problems with these mechanisms is striking the balance between having a short timeout period for those that require high assurance, and overloading the system with the CA to directory traffic, requiring the CA to move closer towards an on-line mechanism.

The main feature of our amendment, is the ability for the user to require a call-back mechanism if so desired. What this amounts to, is the user having the ability to retrospectively receive a finer degree of granularity to the information provided by the directory. If the existing mechanism used an off-line verification mechanism which has an update period of T, then any client unwilling to accept a high degree of risk, will have to wait until the end of the current period before requesting a fresh version of the revocation material.

Under this system a user can only receive information with a granularity of T. Our adaption of these mechanisms allow for a degree of granularity as fine as the service is willing to provide (say δt), while still allowing the distribution of the bulk of information between the CA and the Directory to occur with a period of T.

Take an example where A provides a service, and B approaches A with a request for service. A checks the certificate handed to her by B and then calls the distribution server to assure herself that B's key has not been recently revoked. If it is within A's policy to require a finer degree of granularity than that offered by T, and B's request comes toward the end of an existing revocation period, then A has to either accept a level of risk that is in contradiction with her policy, or request B delay the transaction in order to satisfy herself of the validity of B's key.

Under existing systems the CA signs the revocation information for the certificates with the period that the revocation is processed. By changing this model to one where the CA signs each revocation with a timestamp (some multiple of δt, and then signs the distribution of the superset timestamp (some multiple of T), we can allow for users to receive more accurate information about a particular revocation.

5.3.2 The new protocol

If A requires retrospective confirmation of the lack of revocation of B's certificate, then she includes a signal of this request in the initial service application.

The directory then builds an outgoing queue of these requests to be serviced, and at the start of the next period, sends to A a second confirmation or denial of the status of B's certificate. If the result indicates a revocation, the timestamp received by A will have granularity t, which will allow her to make a retrospective decision as whether to remove the result of the B's transaction within the system.

We highlight the mechanism with an example as follows: The CA distributes a list at time t_1, and subsequently receives notification to revoke B's certificate at time t_2 ($t_2 > t_1$). It draws up the revocation information, adding the timestamp t_2 to this information. B approaches A for service at time t_3 ($t_3 > t_2$), and given the revocation information received by A from the directory at this point, A decides to process B's request. A also flags the directory that she wishes to be notified after the next update of any change to B's status during the current period. At time t_4, ($t_4 - t_1 = T$) the CA issues a new set of revocation information to the directories, at this point the directory processes the outgoing information queue and sends A a copy of B's new revocation status. The time-line is clarified in figure 5.1, with the shaded area highlighting the period where B's certificate has been revoked, but that he is still able to perform requests throughout the system with impunity.

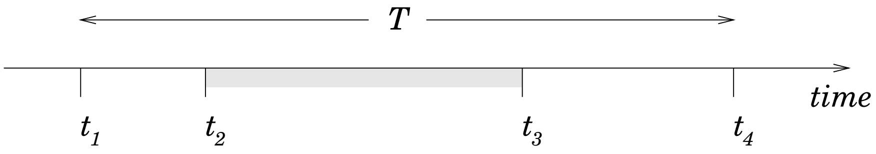

Figure 5.1: A revocation time-line example

We expand on this by describing a set of protocols to implement the scheme. We use the CRL mechanism as our example, for ease of demonstration, although it could be used in conjunction with any of the other, more efficient schemes.

There are three sets of protocols involved here. One protocol layer involves the distribution of the revocation information between the CA and the directories, and a second is the request from B to A. As our mechanism does not differ in these interactions from the standard schemes, we do not give examples of the protocols here. The third protocol layer is the interaction between A and the directory, and is described below.

The protocol makes use of the following information:

- D_i - An instance of the directory.
- CR_j - A certificate of revocation for the public key of the user ID_j. It is a tuple containing t_r (time of revocation, at fine granularity), and ID_j.
- RL_d - A Revocation List, distributed to all D_i at time t_d by the CA. It is a signed set of all current CR_j's and the additional timestamp t_l (time of dissemination of the list), in multiples of period T.
- F - A flag, used by A to communicate to D_i the desire for a revocation call-back on B's status.
- t_{cr} - The time of the issue of the request, this is only included if the flag is set to require a call back.

The protocol itself is shown below, with A communicating with some D_i:

$(1) A \rightarrow D_i : \{A, B, F, t_{cr}\}_{K_A^{-1}}$
$(2) D_i \rightarrow A : \{A, B, RL_d\}_{K_{D_i}^{-1}}$
$(3) D_i \rightarrow A : \{A, B, RL_{d+1}\}_{K_{D_i}^{-1}}$ - dependent on F

The third message of the protocol is sent if the value of F is set in the first message from A. This message is sent regardless of whether B's certificate had been revoked between time t_d and t_{d+1}. This allows A to receive both the negative and positive acknowledgement of the action.

If A does not receive the third message from D_i, at time t_{d+1}, after requesting it in the first message (this could be due to D_i crashing in the meantime, or D_i maliciously not replying to A), then A can issue another one off request to some other instance of the directory server D_l ($l \neq i$), using the same protocol as above, and not requesting a call-back. This works, as A receives RL_{d+1} in both cases.

To make use of this revocation mechanism, it requires the environment to have the following properties (the first is necessary throughout the whole environment, but the second need only apply to servers who wish to use the callback mechanism):

1. **Loose Clock Synchronisation** - This property need not hold to very tight bounds, but for A to make a decision with regard to the validity of B's certificate, by comparing t_{cr} with t_r, there should be some degree synchronisation within the system.
2. **Negatable Transactions** - It is the nature of our extension that it is of use to clients that run a system that allow for reversible transactions. That is, all transactions are negatable over a set of input transactions in a manner that the transactions can be re-done in order to eliminate the effect of one of the transactions. This might seem like a heavy requirement, but can be achieved in some services. Two systems that can be engineered to meet the requirement are:
   (a) An electronic money transfer system, where the system can inject the inverse transaction to balance the transaction[1]
   (b) A file system or database that allows for a generation of a log file associated with transactions in order to remove the effects of a change to the data. All transactions can be finally committed at the end of the current time period, and the log files dropped after A is satisfied that all transactions are valid.

We observe that our systems has a parallel in access control, notably that of optimistic access control, with a post-hoc freshness certificate [2]. Although there is a difference in the push vs. pull strategy employed.

5.3.3 Comparison

In our proposed protocol, we aim to tackle the problem of balancing on-line and off-line solutions to revocation. We now compare our method to others proposed, notably those put forward in the context of CRLs in the N.I.S.T. Public Key Infrastructure [U.S95].

The first approach they note is using *Broadcast Revocation Lists*, where the CA broadcasts the lists to certificate using systems (cutting out the use of the directory system altogether). This allows the CA to broadcast on a finer timescale, lists for revocations considered more important. In this scenario it requires a high degree of resilience from the CA, pushes the system towards a virtual on-line system, and forces a high communication

[1] We note that this might sound like a strange design principle, but if we compare it to current practice in a credit card system, where a card user is not charged for transactions after notification of card theft, even when the revocation mechanism has not caught up with the card revocation.

[2] We thank Bruce Christianson for brining this to our attention.

cost directly from the CA. A notable difference between this and our design is that the CA makes the decision on what are considered to be important keys. In the case of a user offering a service, the decision on the relative importance of a key should be met by the local security policy, as it is likely to involve decisions regarding transaction information as much as key material, and our mechanism allows for this.

They also mention two methods of updating their existing directory style revocation structure. In *Directory Updating*, the approach is to have the CA immediately remove the certificate from the directory when a revocation is performed, doing this adds the overhead of securing the communications between the CA and the directory. With *Trusted Directory*, *directory updating* is augmented by having the directory system become a trusted component of the system.

The second of these is clearly less desirable, as a widely distributed directory becomes a bigger target, and moves away from one of the reason off-line systems were initially proposed, to remove the need to trust the distribution mechanism. The first proposal increases the cost in terms of computation and communication between the CA and the directory.

Our proposal does not require the directory to be trusted, this is a clear advantage over *trusted directory*. *Directory updating* pushes the overhead in cost into the CA to directory communication, while our protocol adds overhead to the user to directory communication. One scenario where our system is more efficient than theirs, is where T is relatively large in comparison to the transaction ratio on a client to service basis. For example, if B were to conduct several requests with A in any period between CRL updates, their mechanism would require A to communicate with the directory at every occasion, in our mechanism a single communication with the directory is all that is requested, as all subsequent suspected revocations can be handled by the call-back issued for the first transaction.

5.4 Communication across interfaces

In this section we study the problem inherent in communication across interfaces that supply fault tolerance, and how that communication should be regulated when it comes to security. We illustrate our discussion by using an electronic commerce scenario. Our discussion demonstrates that an analysis of the effects have to be taken into account, and sometimes a change in the model structure is needed when it comes to placing the model within a security environment.

In a system with an interface I, which provides a service, the interface is deemed to provide fault tolerance if, by providing a service to a client C for the objects maintained by the interface, there are some recovery mechanisms associated with those objects [ALS78]. This is represented by the diagram in figure 5.2.

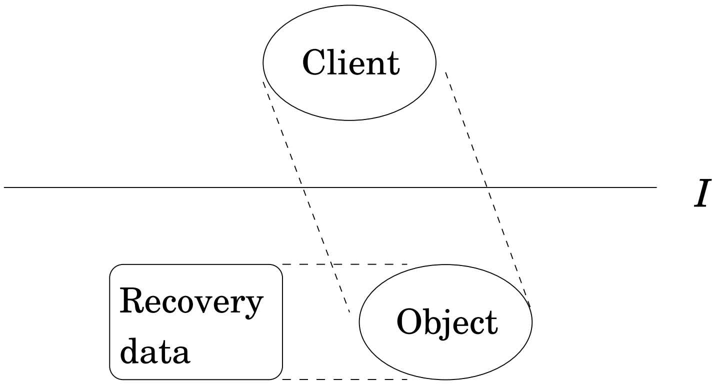

Figure 5.2: Fault Tolerance across an interface

The interesting element of this discussion from a security standpoint is the communication model they express for the Interprocess Communication (I.P.C.) external to the interface (i.e., two processes A and B whom are both clients of the interface I).

In such a scenario they treat processes as either *Competing* or *Cooperating*. It is the difference held with regard to the system at this level that we are interested in.

Anderson and Lee [AL81, pages 208–223] and Anderson et al. [ALS78] describe the differences in the interface's external information flow as follows:

Competing - Competing processes use the same interface, but have no communication that affects the objects maintained by that interface. Hence their view of the use and recovery of any objects maintained by the interface can be regarded as private, though they were the sole user of the interface. Thus if one process has to invoke recovery on an object across an interface there is no need to share such information with other processes that also use the interface.

Cooperating This implies the communication of information through the use of shared objects across the interface directly for IPC. In this scenario, there is a need for information exchange about the recovery of objects maintained by the interface (i.e., if two processes use an interface that provides recovery in a cooperating manner, then there is a need for information flow between the processes with regard to the recovery of shared objects). A problem encountered with recovery for competing processes – which we will not contemplate further here – is the *domino effect* [AL81], where information flow between processes that do not have a coordinated recovery strategy can end in system state rolling back excessively.

When we come to applying fault tolerance to a security interface, we argue that there are environments where these conventions need to be changed. In a secure environment, there can be a compelling need for clients who are not deemed to be cooperating processes – by the above definition – to be able to share information.

We do note however, that there exist other scenarios, where the above model not only needs to be adhered to, but possible tightened – such as a Multi-level secure environment, where information flow between users at different levels (Top Secret, Secret etc.) is undesirable, regardless of their status as either competing or cooperating.

To provide some ground for belief in our argument, we focus on the case of electronic commerce, by looking at an electronic cash mechanism. In an electronic cash mechanism, having some form of fault tolerance is desirable[3].

Yahalom [Yah98] makes a compelling case for having a recoverable mechanism on a security service, under the assumption that perfect security is unattainable, having a mechanism that can catch cheaters after the fact it is just as desirable.

We look at such a principle applied to an electronic commerce scenario[4], with the following players:

$\mathcal{C}$ - A set of clients (where $A, B \in \mathcal{C}$).

$\mathcal{D}$ - A set of n dealers (D_i, $i = 1 \ldots n$), providing a digital cash function, which members of $\mathcal{C}$ use in a client-server environment[5].

M_j - A set of p merchants ($j = 1 \ldots p$), where members of $\mathcal{C}$ can spend the digital money[6]

[3] If we look at the example of Mondex [SC98, Ber96], they have a backup transaction protocol which can be deployed if the first is broken.

[4] Although we do not discuss a particular implementation.

[5] We assume that members of $\mathcal{C}$ are free to chose any D_i and can change easily between D_i and D_k ($i \neq k$).

[6] For simplicity we assume that members of $\mathcal{C}$ only spend money with any M_j and do not interchange with each other. This helps keep the model strict to the definition held for competing processes.

The reason for restricting the digital cash mechanism to clients, is to adhere directly to the principle of competing processes, which helps simplify our argument.

We require that the environment the clients operate in allow free movement of the clients between potential dealers, otherwise there is no recourse within the system, and a client, once having chosen a dealer, has no method to change dealers if they are unhappy with the service. This is a realistic assumption, but one that needs to be made explicit to enable our discussion.

Each of the dealers provide a digital cash mechanism, with the added functionality of a recovery mechanism similar to that proposed by Yahalom. A client which notices an error in the value of their digital cash will invoke the recovery mechanism. In order to do this they will have collected the relevant evidence they require to prove to their dealer D_i that they have been defrauded.

In a scenario where both A and B bank with D_i, if A suddenly notices she is having to use the recovery mechanism built in to D_i's cash system more and more often, then she is likely to want to move to another dealer D_k because of the inherent risk involved with banking with a dealer where recovery is consistently required. Because A does not cooperate with B across the interface provided by D_i, by using the recovery mechanism, there is no means by which they can share information regarding the objects maintained by D_i. In a security environment, where the client requires a high degree of integrity from the interface, A may wish to provide B with evidence of the recovery of her object on D_i's interface, which influenced her subsequent decision to move to dealer D_k.

B might not necessarily take notice of A's comments, which comes down to B's security policy, and there clearly needs to be a trust relationship between A and B for such action to be taken.

If A does communicate with B with regard to the recovery of objects on a common interface, then she is going to have to provide evidence to B of such actions for B to be collected.

In Yahalom's work, there is a requirement for evidence upon which B can make his own decision before recovery can take place, such evidence can also be used in the communication with B, but this requires careful design of the evidence for B to be able to make use of such information with regard to his policy decision[7].

It is important that such an instrument does not become open to abuse

[7] If we take the earlier example of a multi-level secure system, where communication might be undesirable, the converse is desirable. A should not be able to gather conclusive evidence that she could present to B with regard to any recovery across the interface.

by a client, otherwise a malicious client could generate false evidence that objects maintained by D_i were subject to recovery. This could cause D_i to lose business, in a form of denial of service attack.

5.5 Disjoint & Inclusive recovery

Continuing with the work from the previous section, the model of recovery across an interface is expanded upon in the fault tolerant literature [ALS78]. The expansion is an extension of the simple recovery across a single interface environment to an environment that has multiple interfaces. The term used in the literature for such a design of this type is a *multilevel system* [ALS78][8], and is split into two categories. The first is an *interpretive multilevel system* and the second an *extended interpreter multilevel system*. The difference between them is, that in the first, the abstractions provided between an interface and the processes that use the interface are clean, in that the process has no notion of the provision of those services provided below the interface it sees. In the second case, the process has access both to the extension and to the original interface.

It is this second concept of an extension on an original interface and the different types of fault tolerance it provides which interests us here.

Anderson et al. [ALS78] describe two types of recovery scheme for this environment. The first is *disjoint recovery* an the second is *inclusive recovery*. The main difference between the schemes is the manner in which the calling process is notified of the recovery. As an example, if we take process P, calling an extension E to an interpreter I with object O, in the *disjoint* scheme, E is expected to take care of any recovery on the data it holds internally with regard to O separate from the recovery data that I holds on O. Thus, if P requires a recovery operation on O, E is expected to do its own internal recovery with regard to this data. In the *inclusive* scheme, all the recovery information E requires is also maintained by I, thus when P recovers O, E does not have to carry out separate recovery on any internal information.

The key to these differences is the transparency of the recovery across the extension, and it is this transparency and its effect on security that we discuss here.

[8] There is a clash of terminology here between a multilevel system in terms of security and in terms of fault tolerance.

5.5.1 In terms of security

We now take these conventions and adapt them to be applied in the context of security mechanisms. Here we re-define these terms for our own use:

Disjoint A disjoint security interface is expected to provide recovery without involvement from any information maintained by the client process. Within the notion of transparency, this type of recovery is taken care of without client intervention.

Inclusive An inclusive security interface is reliant on the client process for some of the context with which to perform the recovery itself. In our view of transparency, this method requires intervention from the client process.

We make a connection here to an argument put forth by Gollman [Gol98], where concern is made over the fact that the client might be facing a different attack threat to that noted in the classical literature. It is also noted by Needham [Nee94] that security protocols are usually deemed to “just work”, without any consideration to the consequence of what should happen if they do not.

The main issue we wish to propose here is, that if a security service is to make use of fault tolerant recovery primitives (using recovery across an interface as our model for this), then the type of error recovery needs to be taken into account. We expand on this below:

The application of the type of recovery will be affected by two choices. Firstly, what security functionality is the system delivering, (e.g., secrecy, integrity etc.). Secondly, where does the trust boundary of the client using the service lie.

Taking these in turn we discuss the implications on the choice of strategy used.

Secrecy

Because of the very nature of secrecy, it is impossible for the interface providing the service to recover without aid from the calling process. Thus, using an inclusive recovery strategy is required for any interface that provides secrecy.

Although an interface which provides secrecy must use an inclusive strategy, there is one scenario where a disjoint recovery mechanism can function, and that is for the secrecy of keys. It is feasible that, if an interface provides a key management service, then re-keying on behalf of the calling process can be done separately. Essentially this is what a session key set-up mechanism is doing. In the example where a session key is used for a long period of time, re-keying is already recommended practice, to reduce the possibility of some forms of cryptanalytic attack. This is an example of a scenario where a disjoint mechanism can be recommended, although notification to the client of re-keying would be advisable in the case of recovery.

Integrity

Here we will differentiate between what we regard as two types of integrity:

Absolute Integrity in the classical sense, where the data transmitted or stored can be checked for tampering in transit or storage.

Relative Integrity of the process or procedure. This is more amorphous, dealing with possible subjective context.

In the case of absolute integrity, there is a case to be made for the recovery to be disjoint. This would allow for the recovery to be carried out by the extension without the process needing to notice. Such a case would be the re-sending of a secure data packet whose MAC had failed. Such a recovery could be quietly handled without interference from the process. Although the service might wish to notify the client in the case of recovery taking place, but the client does not get involved in the actual process.

In the case of relative integrity, the recovery should be inclusive. This is because of the subjective nature of the loss of relative integrity[9], it is necessary that the process is able act according to its own security policy.

Denial of Service

A security service that aims to provide resilience to a denial of service attack will, by its very nature be designed to handle recovery in a disjoint manner. This will allow the client to transparently use the recovery mechanism of the service, which is its basic provision.

There is a counter argument to this, where a security service that provides for denial of service, but which is ultimately unable to provide those goals should switch over to an inclusive recovery mechanism that allows the process to try and handle the situation as best possible.

[9] To give an example, we view our work in chapter 8 as being an example of this form of integrity.

5.5.2 Policy related recovery

After noting which types of security mechanism the different conventions are best tailored to, we note that a primary motivator can also be the security policy of the client, and the amount of information that an interface will supply to the client when it interacts with the service across an interface.

We illustrate our argument with a short example to highlight the potential for conflict:

If a security interface is going to provide a service for a client, it may keep information regarding the service it provides separate from the information it shares with the calling process, even though the information is linked. In this type of scenario, it makes sense for the recovery mechanism to be implemented as disjoint[10].

On the other hand, it could be argued that an interface that provides a security mechanism might be better served by allowing the client that uses the interface to be involved in the recovery process (i.e., inclusive recovery). This would then allow the process to use its security policy as an input to the recovery process, providing an open environment, and allow the process to have more faith in the recovery mechanism.

Although we only touch the description here briefly, we believe the use of differing recovery policies across different types of interfaces to be a useful abstraction that could provide for some interesting research in future.

5.6 Conclusions

Our goal in this chapter was to look at some of the mechanisms used within the fault tolerant literature, adapt their use for secure services, and note how, in crossing from one domain to the other their bounds needed to change, and how we can use their goals to provide empowerment within services of a secure nature.

Firstly, we adapted the notion of raising exceptions for use in a revocation mechanism. We find that given some constraints with regard to their use (constraints which are inherent to the mechanism itself), we are able to augment the functionality of revocation service. Particularly, we are able to balance the benefits and drawbacks of off-line and on-line recovery mechanisms – namely granularity v. timeliness. We find the notion of exceptions fit neatly into the model without need for adaption.

[10] We note the similarity here to the information hiding principles underlying Object-Oriented programming.

Secondly, we studied the notion of information flow external to an interface. Such information flow is regulated in its original environment by the definition of cooperation, specifically with regard to its use within IPC across an interface. We conclude that this model needs to be changed when adopted within a secure service. We present evidence for our claim by providing an example where communication external to the interface, directly related to the service provided by the interface, can be beneficial to the client. Although we do note, that such changes are not necessary or desirable in all environments (e.g., multi-level secure systems).

We finish by looking at the notion of disjoint and inclusive error recovery across and interface. Here we see that the model conveniently translates into its new environment, but that we need to provide more emphasis with regard to the motivation for its use – something that is given little thought in its original environment. If we concentrate on our original premise of reduction in trust, we conclude that the use of an inclusive recovery scheme in such scenarios can be beneficial.

Chapter 6

Resilience to timing attacks

6.1 Overview

One of the many types of replication checks mentioned in the literature are *timing checks* [AL81, pages 123–124]. They are a limited form of replication check that can be applied if the specification of the component provides for some form of timing constraint. They are quite flexible in that they can be used in both software and hardware environments (e.g., replicated hardware processors can ensure the availability of other processors, and watchdog timers can be used in software to catch code segments that could have fallen into an infinite loop).

In this chapter we describe a simple method of using Asynchronous Transfer Mode (ATM) network technology to defeat attacks that rely on the opponent's ability to disrupt the timely delivery of messages within a cryptographic protocol. Our method centres on ATM technology's ability to provide guarantees associated with the bandwidth and delay characteristics over a connection. We manipulate these mechanisms to provide us with timing checks on the delivery of cryptographic protocol messages, which can be used to monitor for foul play in the message delivery process. We also describe how this can be used to detect a denial of service attack. ATM Quality of Service (QoS) is briefly described, along with a functional breakdown of a proposed design.

6.2 Introduction

This chapter describes a method of using Asynchronous Transfer Mode (ATM) networks to defeat or detect various timing or replay attacks on cryptographic protocols.

Attacks on cryptographic protocols that use methods of disruption within the timing of message delivery within a given protocol run are well known in the literature. Work by various authors [Gon92, Gon93b, Syv94, LG90, Aur97, Mit89, OP96] show the potential for such attacks on certain protocols. Each of these attacks has its different form, requiring a differing amount of knowledge about the messages and their content, in order to be successful. What these techniques have in common is that they disrupt the flow of the messages within the protocol run to some degree. Gong [Gon92] makes clear that the main reason this is possible is down to the inherently unreliable networks that are used in the majority of distributed environments. Networks that employ standards such as Ethernet – due to their broadcast nature – are susceptible to dropping packets if the traffic is busy. This means that if a given packet is lost on such a network, the act is unlikely to be noted with any interest. This allows an attacker to interfere with packet transmission with impunity.

O'Connell and Patel [OP96] give a model whereby the trip latency on messages between two hosts is measured, this is then used to try and detect message delay. Their work – although not requiring synchronised clocks – relies on the measuring of real time, and reasonable measurement of clock drift between the two hosts. A delay is calculated by a measurement from when the start of the message is sent to when the end of the message is received.

Gong [Gon92] also gives a clear example of how even a thoroughly studied protocol such as the Kerberos authentication mechanism can fall foul to such attacks under circumstances which are difficult to predict.

By removing this unpredictability from the underlying communications infrastructure, we are able to reduce an attacker's ability to launch such attacks, also allowing a legitimate client to detect and respond to a potential attack.

We use ATM as a network architecture to counter this problem because of the bandwidth guarantee that can be obtained on a connection. This design principle was introduced into ATM in order to service applications that needed regular bandwidth (such as video phones). We hijack this for our own ends in order to better regulate the passing of messages between principals that take part in a cryptographic protocol.

The rest of this chapter is organised as follows – section 6.3 gives a better overview of the bandwidth regulation methods implemented in ATM and how we can use this to our advantage. Section 6.4 goes on to give an example of how we could implement our proposal within the protocol stack at a host, providing service to applications that use cryptographic protocols. Section 6.5 gives some examples of its use against attacks noted in the literature, followed by our conclusions in section 6.6.

6.3 ATM Functionality

In this section we look at the functionality that an ATM [LM95] network gives the operating system and applications that run on that network. We then study this service from a security point of view, showing how we can exploit its ability to give us guarantees on the delivery of cryptographic messages. We end the section with two features of ATM (*cell delay variation* and *discard control*) and discuss how they specifically affect our work.

6.3.1 Service guarantees and QoS

It is a feature of ATM networks that they allow a connection (called a Virtual Circuit or V.C.) to have certain upper and lower bound characteristics with regard to bandwidth and delay. These mechanisms were placed into the network architecture in order to give users of time critical applications some degree of protection against network delays. Combined with the application level use of these features, this is generally known as Quality of Service (QoS).

If an application is communicating via an ATM network, it can request guaranteed bandwidth and delay on a V.C. that it wishes to set up. The network then either accepts or rejects the request for the connection, depending on the resources requested when balanced against current resource allocation.

The ATM standards were – in part – designed to deal with shared media over LAN's that require low connection costs. It is the characteristics of the type of connectivity required in these environments (i.e., low bandwidth users and infrequent users) that we exploit in our aim to defeat timing attacks.

6.3.2 Bandwidth division

Not all V.C.s are given guaranteed bandwidth on an ATM network. This allows us to effectively segregate the bandwidth on a network into two components.

By giving an application that requires the use of a secure protocol the ability to request a V.C. with associated guarantees on bandwidth and delay, we can accurately predict the delay experienced by any message sent. Any deviation from this delay indicates the possibility of an attack. This does not always indicate an active attack – as no guarantee is absolute – but we can make the probability arbitrarily high.

We abstract this as dedicating a portion of the bandwidth for cryptographic protocols. Because of the nature of the majority of cryptographic protocols – small number of messages, with little bandwidth consumption and infrequent – they fit exactly into the category of application that ATM QoS was designed to support.

This division should not disrupt the throughput for the majority of traffic, as once the cryptographic protocol is complete, the guaranteed bandwidth held by the principal is dropped.

6.3.3 Separation of function

It is a factor of other network designs that all communications on the network suffer under high load. By taking advantage of a channel which has its bandwidth and delay characteristics regulated, we are able to stop many timing attacks. In essence, we are separating the communications infrastructure into different portions – separate *secure* and *insecure* channels – each of which provides us with different functionality. Making use of this *secure* channel does not address any other issues in terms of security; it does not stop eavesdropping, and all checks for integrity and authenticity are still be carried out by the cryptographic functions used by the application.

6.3.4 Cell Delay Variation and Discard Control

Information is physically sent using *cells*, the information distribution packet used on a V.C. Because of the queueing involved along the route that a cell travels from source to destination, the cells suffer from what is termed "Cell Delay Variation" (CDV, also called *jitter*) [GB94]. On a "Constant Bit Rate" (CBR) connection – which we use for the *secure* connection – the cells are sent out at regular intervals. Some cells will experience a shorter delay than preceding ones (causing *clumping*), and others will experience larger delay (causing *dispersion*). It is the statistical representation of these variations from cell to cell that defines the jitter. As the jitter on a given channel is likely to change gradually through time, any radical change in CDV around a *secure* connection will indicate the likelihood of an attack to the receiving end.

An advantage of measuring CDV over straight message delay is that cells should arrive at a roughly even distribution, subsequently any massive clumping (e.g., the whole message in successive cells) would signal an attack.

This forces an attacker to carry out an attack without knowing the message content, because of the need to not disrupt the delay characteristics.

There is also a feature called *Discard Control* [LM95], which enables the network nodes, in the face of severe congestion, to drop cells from the same packet, as dropping random cells that come from different packets causes greater disruption for the overall delay characteristics of the network. Queueing mechanisms use priorities to decide which cells to pick when transferring cells from incoming to outgoing queues (thus affecting which cells might get dropped). With some implementations, the application can allocate a priority level to both delay and loss characteristics [CU95]. We would need to allocate the highest possible priority to each of these to minimise any disruption to our *secure* connection. The ideal circumstance would be to have a priority mechanism which would ensure that *secure* V.C.s were not dropped under any circumstance – although we are not aware of any ATM queueing mechanism that would currently allow this.

6.4 Changing the protocol stack

A simple representation of the protocol stack used in ATM is shown in figure 6.1 (from [LM95]). The proposed inclusion of a secure layer implementing the desired functionality has been added to the diagram. This could either be a new layer or be incorporated into an existing layer. For the purpose of our discussion, we will assume it resides above the AAL-x layer.

We describe an *initiator* of a secure V.C. to be the principal that requests the creation of a V.C. in order to transmit a cryptographic message[1]. A *receiver* is the recipient of the message.

The ability to be both an *initiator* and *receiver* of a secure connection should be available to both client and server applications. This is necessary because several authentication protocols (e.g., Needham-Schroeder [NS78]) have messages where both *initiator* and *receiver* are clients. Although we are not aware of protocols where servers initiate the cryptographic protocol, they frequently initiate a subsection of the protocol, sending a message to the second client in an authentication protocol, after being contacted by the first client (e.g., Wide-Mouth Frog [BAN89]).

We restrict the number of *secure* V.C.s that are allowed to be open at any one time at a given host. This gives us an upper bound on the round trip

[1] We use the term cryptographic message to describe a single message sent from one principal to another, that is part of a larger protocol (e.g., $[A \rightarrow S : A, B, N_a]$ constitutes a single message).

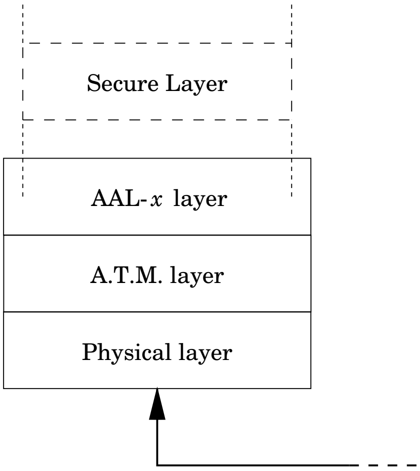

Figure 6.1: Current (and proposed) ATM layers in stack

delay for a message and its reply, because processing time at the receiving end will be – amongst other things – a function of the number of connections open. This is required in order to ensure that we are able to monitor the connection accurately. This should not cause a problem as far as requesting a secure connection is concerned, as cryptographic protocols are lightweight in terms of modern communication and their time to completion is likely to be quite small. Subsequently, a full input queue will be cleared quickly.

The required functionality this layer should exhibit is characterised below:

1. Initiator Side

   - Negotiate a secure V.C. from *initiator* to *receiver* with the underlying AAL-x layer.
   - Notify the initiator of the connection time on a V.C. carrying a message that has no response (such as the final message in the protocol), and whether this delay is acceptable.
   - Notify the client of the round trip delay of a message that is followed by a response (this includes both the time spent in both in-bound and out-bound connections and the processing time for the server). A breakdown of V.C. connection time and processing time should also be available.
   - If a request for a secure connection is refused, then back off and re-issue the request. This should be carried out a pre-set number of times before giving up and signalling an exception to the calling application.

2. Receiver Side

    - Notify the *receiver* process that an incoming V.C. is a secure request.
    - Notify the *receiver* if the delay on the incoming message falls outside the negotiated window.
    - Negotiate the response V.C. on a message in a protocol that requires a response (e.g., the second message in a handshake).
    - Notify the underlying AAL-x layer if the receiver already has the maximum number of V.C.s open, in order that a request for a new *secure* V.C. should be denied.

When negotiating for a *secure* connection, all parameters should be within a constant window (i.e., for a given network, all *secure* connections will have bandwidth and delay characteristics within a pre-defined envelope). This ensures that all participants in a cryptographic protocol are able to judge the outcome of their requests accurately. It would be ideal to have these characteristics fixed, but on longer routes through a network it would be implausible to guarantee the negotiation of discrete amounts.

Lam and Beth [LB92] state that a server should not hold any state if possible [2]. This is not of a great concern in our design for two reasons:

1. A server is limited by the number of connections it has open at any particular time.
2. As the client is the *initiator* in the majority of cases, the client will hold the state, reducing the burden of storage on the server, which is also desirable from the point of view that we aim for the clients to be in control of the protocol.

6.4.1 Function calls

Figure 6.2 shows an enumerated set of function calls to implement the scheme, with their descriptions given below:

1. Initiator calls

    1 : open_link This is a call made by the application to the secure layer, it includes the address of the server to send the request to and includes the message content.

[2] We look further at the relevance of the state that a server maintains in section 7.4.

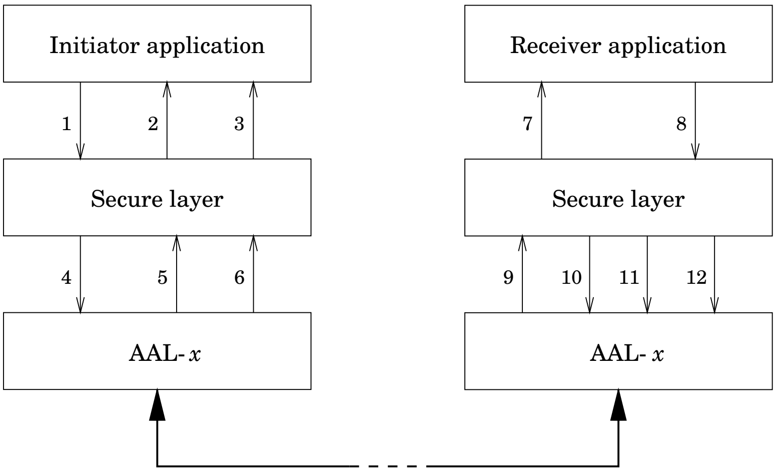

Figure 6.2: Enumerated diagram for secure layer function calls

2 : close_link This is the successful request return made by an application. It includes a flag of whether the secure layer believes the request has been delayed or not.

3 : link_fail Informs the client that a connection for a V.C. was denied more times than a pre-set threshold.

4 : link_req The secure layer uses this to request a V.C. with the required bandwidth.

5 : ref_req A return signal to the secure layer indicating that the request has been refused.

6 : return_msg This is a return message from the *receiver* to the *initiator* (if a response is part of the protocol). This is passed up to the application as a return message.

2. Receiver calls

7 : deliver_msg Delivery of the message body from the *initiator* and flagged to indicate that it is a secure V.C. and if the delay is acceptable for the negotiated connection.

8 : msg_out Outgoing message from the *receiver* to *initiator.*

9 : conn_req Signal for a request of a secure V.C. from an *initiator.*

10 : conn_ref Return signal refusing the request for a secure V.C. (due to a full incoming queue).

11 : conn_acc Return signal accepting the request for a secure V.C.

12 : return_msg Passing on the outgoing message from the server to the client.

When the secure layer receives a request from an application, it needs to negotiate a *secure* V.C. with the AAL-x layer. If the connection is accepted, then the secure layer starts a timer until the connection is closed by the *receiver* end of the V.C. It then checks if the actual connection time falls within the expected limits, flagging the result to the application.

Given that an attacker might try and give a false message to the *receiver* before the *initiator* sends the message, it should be up to the secure layer to wait for the whole duration that the V.C. is expected to be open before passing the message up to the application layer. As the *secure* layer cannot distinguish the difference – because any authentication is done at the application level – all messages must be passed up to the application. Although the arrival of more than one message on a secure V.C. is itself likely to signal an attack.

6.5 Examples of usage

We now take a look at both replay attacks and denial of service attacks, evaluating how our scheme helps to prevent and detect these types of attacks.

6.5.1 Timing and replay attacks

First of all we take a look at some of the replay attacks described by Syverson [Syv94]. In his paper he describes two separate taxonomies, an *Origination* taxonomy and a *Destination* taxonomy. Each of these is split into subsections as follows:

- Origination taxonomy
    1. Run external attacks (replay of messages from outside the current run of the protocol)
        (a) Interleavings (requiring contemporaneous protocol runs)
        (b) Classic replays (runs need not be contemporaneous)
    2. Run internal attacks (replay messages from inside the current run of the protocol)
- Destination taxonomy
    1. Deflections (message is directed to other than the intended recipient)

   (a) Reflections (message is sent back to sender)
   (b) Deflections to a third party

2. Straight replays (intended principal receives message, but message is delayed)

We now look at each of the different threats noted above, and discuss the usefulness of our technique at avoiding and/or detecting each of them in turn.

Taking the *interleaving attack* described by Syverson (the BAN-Yahalom protocol [BAN89] and the attack on it are shown below), it is possible to see that A might raise the alarm in this situation. All principals involved in a cryptographic protocol can assume that all messages on a negotiated *secure* V.C. are sent and received. When A does not receive the message in round 3 of the BAN-Yahalom protocol, it is difficult for the attacker to cover the attack. If the attacker does not tear down the legitimate protocol, then B ends up receiving two versions of message 4 (one from A and one from ${E_a}^3$)

The BAN-Yahalom Protocol
$(1) A \rightarrow B : A, N_a$
$(2) B \rightarrow S : B, N_b, \{A, N_a\}_{K_{bs}}$
$(3) S \rightarrow A : N_b, \{B, K_{ab}, N_a\}_{K_{as}}, \{A, K_{ab}, N_b\}_{K_{bs}}$
$(4) A \rightarrow B : \{A, K_{ab}, N_b\}_{K_{bs}}, \{N_b\}_{K_{ab}}$

Attack on the BAN-Yahalom Protocol
$(1) A \rightarrow B : A, N_a$
$(2) B \rightarrow S : B, N_b, \{A, N_a\}_{K_{bs}}$

$(1') E_a \rightarrow B : A, (N_a, N_b)$
$(2') B \rightarrow E_s : B, N_b', \{A, N_a, N_b\}_{K_{bs}}$

(3) Omitted.
$(4) E_a \rightarrow B : \{A, N_a(= K_{ab}), N_b\}_{K_{bs}}, \{N_b\}_{K_{ab}}$

Our system cannot defend against *classic replays* of the sort described by Denning and Sacco [DS81]. Their attack is mounted by a masquerader gaining access to an old session key, then initiating a fresh connection using a replay of part of the protocol in which that key was issued. We cannot

[3] Where E_a is used to denote E masquerading as A.

defend against this type of attack, primarily because of the off-line nature of the attack [4]. Although our design could be used to augment systems such as those described by Lomas et al. [LGSN89] that do defend against such attacks.

In the case of *run internal attacks*, we have a similar defense to the interleaving attack described above. Due to the assurance that the principal has of the delivery of messages, it can regard a breakdown of a protocol run as suspicious.

Reflections and *deflections* can be detected if the attacker has to falsify a dropped protocol to interrupt the connection, similar to the case for interleaving.

In the case of *straight replays*, they are similar in concept to the *suppress-replay* attack discussed by Gong [Gon92], where any message that is delayed is going to be picked up as a potential attack by our design.

An interesting point to note, is that if one connection shows signs of being tampered with, then it could be an indication of that connection being used as an oracle in attack on another connection in the system (as demonstrated from the BAN-Yahalom attack described above). The difficulty rasied by this question is trying to reconcile to two instances, which could be an interesting topic for further research.

6.5.2 Protocol Suitability

We take a brief look at how our scheme lends itself to different structures of protocol design.

$(1) A \rightarrow S : A, B, N_a$
$(2) S \rightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
$(3) A \rightarrow B : \{K_{ab}, A\}_{K_{bs}}$
$(4) B \rightarrow A : \{N_b\}_{K_{ab}}$
$(5) A \rightarrow B : \{N_b - 1\}_{K_{ab}}$

By taking the enumerated description of Needham-Schroeder [NS78] given above, we can see that each client is able to encapsulate messages as pairs to and from another party in this protocol (e.g., A sees messages 1 & 2 as a pair to S). This means that they can make an accurate estimate of the round trip delay of this small part of the protocol that they view. In a protocol where clients are not involved in consecutive rounds of the

[4] It is not really a goal of our system to defend against off-line attacks.

protocol, it is more difficult for them to accurately predict the total delay expected until they receive their next message [5].

If we contrast this simple example to that used in the *interleaving* attack above – where there are no message pairs – we rely to a greater extent on the reliability of the communication in the case of the protocol studied in the *interleaving attack*. We observe that our method is more adept at monitoring a protocol where there is symmetry in the message structure of the protocol (i.e., a protocol where a principal sends a message and receives a response directly from the principal that the first message is sent to). If, as part of a protocol, a client needs to send a message to another client, and does not receive the reply for a number of rounds within the protocol, then some inaccuracy in the calculation of the total round trip delay might be experienced.

6.5.3 Denial of Service attacks

It is possible to use the methods that we describe in order to detect a denial of service attack mounted against a cryptographic protocol. Indeed, in the example given by Needham [Nee94] of the vault communicating with an alarm company, it is noted that the communication path between the two entities should have a guaranteed bandwidth.

As our design is more general, it is not possible to guarantee bandwidth to all clients at all times, as would be needed in the example of a vault. This leaves us requiring a method to monitor the use of *secure* channels. If the server is refusing another connection (e.g., if its input queue was already full) then as noted previously, the secure layer in the protocol stack would back-off for a short period, then re-issue the request. Because of the infrequency with which most cryptographic protocols are used, it is reasonably safe to assume that a server is unlikely to have a full queue for long. If the client end has to back-off more than a certain threshold, it would be reasonable to assume that the client or server was coming under a denial of service attack. The secure layer would then stop trying to establish the V.C., signaling an exception to the application. The question of whether this was a denial of service attack, or some more innocuous event could be answered by some out of band means.

[5] It could be a policy that all connections through intermediaries are delayed for a predefined time (which would be a function of maximum message delays on the network), which would have the effect of re-synchronising the protocol at each step, although this does leave itself open to hijacking if the intermediaries are not fully trusted.

6.6 Conclusions

Timing and replay attacks are well noted in the literature. Other attempts have been made to remove the possibility of replay attacks, most are centred around designing protocol messages in such a way as to try and prevent rounds of a protocol from being forged. In this chapter we present a complementary method of countering these attacks, by using the timing of the connecting channel between the principals involved in the protocol.

We view our design as augmenting current mechanisms for defending against timing attacks. In contrast to other work, ours does provide a mechanism for detecting a denial of service attack which – with the notable exception of work by Needham [Nee94] – has little coverage in the literature.

Our work has two distinct advantages when compared to that of O'Connell and Patel [OP96]. Firstly, continued delivery can be measured by using CDV to monitor cell by cell arrival, which has a distinct advantage over single message timing. Also, messages are guaranteed to be able to get through within a short time, independent of bandwidth usage by other applications for *non-secure* V.C.s.

By requesting a V.C. with a given delay and bandwidth characteristics, and then measurng the variance within the performance received, it is possible to monitor – to a high degree of probability – any interference in the delivery of a protocol message. Any delay would signal the possibility of an attack to the communitating applications.

We analysed how our design would detect the attacks mentioned in a taxonomy by Syverson [Syv94], and show it handles *on-line* attacks (such as an *interleaving attack*) effectively. We also discuss our design's ability to deal with denial of service attacks. We conclude that it would handle this very efficiently. Any denial of service attack mounted against a specific client or server would be difficult to hide, given that our mechanisms monitor requests for service at either end of a V.C. and are not primarily concerned with bandwidth consumption across an entire network.

We do not propose this method as a replacement for thoughtful and correct design of protocols used to implement a secure service, but we do see it as another tool that can be used to improve system security.

Building a working implementation of the *secure* layer would be interesting, primarily to measure the potential occurence of false negatives received under high levels of network load.

We would also consider implementing a new queueing mechanism. It would have a priority level for cryptographic protocols in order that they would not be dropped under any circumstances, and given the highest priority through switching mechanisms. This should remove any possibility of a false negative.

Policing of this mechanism for any abuse by applications would have to be a concern. The security module within each host on the network could be used to monitor for suspicious behaviour from any given application, raising an alarm with the network manager if need be, although pursiuing this more fully would provide interesting research.

Further work could be oriented towards the precise characterization of the properties required by our design. These definitions could then be used in the implementation of future networking standards, rather than shoehorning our requirements into exisiting mechanisms. It is a feature of most implementations that they are designed with inadequate security features in mind, and a set of clear design goals for channel security could be one method of changing this.

To highlight the potential usefullness of our mechanism, we outline two scenarios where we believe our mechanism could be used effectively.

The first scenario is where maintaining the physical security of the LAN could be prohibitively expensive. Our mechanism could provide for a cheap and cheerful means of monitoring for potenial foul play on the network.

The second scenario is where several LANs are connected via an ATM backbone. The propsal for this type of network topology is increasing [Hal]. The physical security of the two LANs might be easily guaranteed, but the backbone is a shared media with other untrusted entities. This type of topology might be common in a commerical environment where the company has many sites and wishes to provide a company wide network, without incurring the cost of a the dedicated hardware to provide this. Commerical espionage in this environment might provide a lucrative target.

Chapter 7

Using Primary-backup for replication

7.1 Overview

In this chapter we develop the notion of using the Primary-backup approach to replication as an alternative to the state machine approach in replicating secure services. Although less robust than the state machine, we give some scenarios where the more lightweight replication mechanism is better suited to the task.

7.2 Introduction

In this chapter we use the Primary-backup approach as a method of providing replication within secure services. It is seen as a different method of developing replication services to the state machine approach. In their article, Guerraoui and Schiper [GS97] comment on the different types of replication techniques for replicating servers – Primary-backup [BMST93] and State-machine [Sch90] (termed Active replication in their paper). The State-machine approach has already been used to develop replication methods for secure servers [Rei96a]. While these techniques may be appropriate for services that require an extremely high degree of robustness, we show that by using the Primary-backup method, we are able to provide a service that gives less rigorous guarantees in scenarios where there might be a need for some lightweight replication.

The argument for relaxing the security requirements in order to provide a more lightweight mechanism has already been put forward in the literature by Syverson [Syv97]. His argument centres around the fact that not all scenarios in security take on the case where Byzantine failures [LSP82] occur, and that by relaxing the need for tolerating such failures, we can subsequently improve the efficiency of failure tolerant mechanisms in security.

We organise the rest of this chapter as follows – section 7.3 gives an overview of the Primary-backup approach to replication. In section 7.4 we show the basic modifications we need to make in order to apply such techniques to secure services. Section 7.5 gives our view of the different type of trust scenarios that we envisage adapting the design to, and we then go on in section 7.6 to supply some simplified high level designs for example services, followed by our conclusions in section 7.7.

7.3 A description of Primary-backup

The Primary-backup approach to replication was first discussed in the literature by Alsberg and Day [AD76]. Their work was initially developed for the sharing of resources. They describe a method where there are multiple hosts, with one of the hosts becoming a *primary* host for the service and the others are *backups*. Any of the hosts can serve as a primary, and can take over if the primary is down. The main benefit their strategy has, is a reduction in the number of messages in the replication mechanism.

This work has also been used by various other people since then [BMST92, BMST93, BM92, MHS89, Gif79] with most of the work going into developing optimal approaches to the replication algorithm given the constraints under which the system operates.

Each of these adjustments to the central idea is built around a different scenario and uses a slightly different method of meeting the goals. They also vary in how the methods allocate the primary, the change over policies, and how the response is communicated back to the client.

7.4 Stateless and Semi-stateless servers

The first problem we encounter when applying this paradigm is the requirement to share information between the replicated servers. The cryptographic means by which most protocols are secured do not lend themselves to sharing information between many communicating hosts – i.e., if we use a shared secret key to keep a channel between two communication parties confidential, then it is not reasonable to share the key with other parties.

In order to circumvent this constraint, we need to apply the securing mechanisms in an orthogonal manner to the information being manipulated by the servers in order to provide service to their clients.

In the strictest sense, there should be no shared state between servers that make up our replication set. We define this property below:

Stateless servers A stateless server is one whose replication mechanisms and use of state to provide the functionality of the group service is not shared by other members of the group. This allows them independence from each other, leading to a scenario where a malicious server can reveal all his information to the world without compromising the ability of other group members to continue delivering service.

It is clear to see from the above definition that it is going to be difficult to implement such mechanisms in practice, as the limitation of information flow between servers deprives us of much of the functionality of the backup mechanism.

In order to impose less stringent constraints on the replication mechanism, we define a less stringent property which we term *Semi-stateless servers* which allows limited information sharing between replicas. We define this property below:

Semi-stateless servers A semi-stateless server is one who shares a limited set of information with other servers in the group, and whose shared information is necessary to allow the replication procedure to provide a richer set of operations. This shared information does not extend to the state necessary to carry out the security critical section of the application. This allows a bad player to broadcast all his secrets to the world without compromising the ability of the other replicas to provide a secure service.

We clarify these differences to demonstrate that services can be replicated in a manner that does not jeopardise the service being provided. An instance of this might be a server replicated purely to avoid a denial of service attack, where the threat model does not include the compromise of the server and its secrets, but an attack on the communication infrastructure which can lead to a loss of service.

It is this second weakened definition that we will use in order to produce a new model of replicating security mechanisms.

7.5 Design modification for security

The Primary-backup paradigm itself offers us a starting base from which we develop different classes of replication. These different classes of replication deal with differing trust scenarios and different classes of threats. What they have in common is the notion that one of a group of servers provides service as a *Primary* and other *backups* are ready to take over providing the service if the primary fails.

We order our mechanisms according the failure mode expected, from the most benign failures that end in a denial of service, through friendly backup to mutually distrusting parties.

The following four subsections provide an overview of each of these mechanisms, given constraints of their ability and general design criteria. The section following the overview descriptions looks into example cases of each of the mechanisms.

7.5.1 Denial of Service

The simplest threat scenario we envisage being able to contend with using this mechanism is denial of service. If we trust the replicas not to divulge secret information, we are only interested in surviving a crash of the replica or an attack on the communications mechanism between client and server.

Each of the servers in a group can perform the same function and can take over from the designated primary if it fails.

If the denial of service results from a communication failure between client and primary, then the client can re-transmit the request to one of the backups in order to receive service.

A server that is resilient to a denial of service attack is the closest approximation to traditional Primary-backup as described in the literature.

7.5.2 Friendly Backup

The next scenario we consider is when a primary is dedicated to providing a function to a certain group of users. Each server that comprises the group is providing the same service, but to a different group of clients. Each primary then acts as a backup for a separate group of clients if the primary of that group crashes or is not contactable.

This agreement is a reciprocal between primaries. More than one server can act as a backup for a particular primary. We can then moderate the level of fault tolerance provided by this service depending on the number of backups each primary is allocated.

It is up to the primary of a given group of clients to inform the group members which backups are available to them, should the normal service be unavailable from the primary.

More formally, we define a group of servers P such that for any i, there is a server P_i that acts as a primary for a group of clients G_i. For each i and j, there is a group of servers P_i^j that act as backups for P_i.

The primary goal of this mechanism is to provide resilience at minimal cost. Instead of providing multiple backups for each server which are then redundant during normal operation, we split the overhead between each functioning server. In implementing this mechanism we must be sure that the loss of one primary is not going to cause noticeable adverse affect on the clients whose primary takes on the role of backup. A method of getting round this mechanism would be to spread the client base of a given primary across multiple backups in the case of failure of the primary. Each client would then have an order in which he tried different backups, moving on to the next in the lists in case there was no contact with the allocated backup. By suitably constructing these lists you could distribute the extra load between multiple backups.

7.5.3 Mutual Distrust

In this scenario, we extend the threat model from attacks via denial of service or server crash, to concerns about the potential integrity of the replicas themselves.

The main goal of this type of replication is to provide service if the primary is unavailable, where the client does not trust another server enough to allow it to function as a backup on its own. In our scenario, if the primary is unavailable, then we can transfer control to a group of servers that collectively act as a backup. We thus do not need to assume total integrity of the backups, but can still maintain a continued service, albeit with a reduction in performance.

The idea of switching the service to a more resilient protocol depending on the environmental constraints has already been seen in the literature, where they term it *adaptive fault tolerance* [GGL93, GG94].

If we extend this scenario to one where the primary itself is not trusted, then we can run the primary as a group, and if the group becomes unavailable, then we can move to the backup process group.

Having an untrusted primary and varying the groups depending on whether the primary group is available or not takes us back toward the full state machine approach, therefore we will not discuss this further here [1].

[1] Although we do cover work in this area in chapter 8.

7.5.4 Segregation of duty

We also generalise this approach to provide some separation of duty between the primary and backups. If either turns into a bad player, they fail to compromise the security of the service, but if one player does turn malicious, then this also yields some degradation in performance. This could ultimately lead to a complete service being unavailable. To breach the security, there would need to be collaboration between the servers that are malicious. This particular scenario is impossible to guard against, thus a decrease in overhead by using a primary-backup mechanism against state machine is welcome, this arises from the split in functionality that comes with segregation.

An example of such segregation of duty can be seen in the key certification mechanism proposed by Crispo and Lomas [CL96] and we will not give our own example here.

7.6 Simple examples

We now provide simple outlines of example implementations that we might consider using a Primary-backup method for server replication. The examples we have chosen are two instances of a key server, a notarization server and a recovery cache. We give some consideration as to which of our replication types we wish to use in each case.

7.6.1 Key Server

We elaborate on two different types of key server. We do this in order to give examples of constructing a server of the same functionality under different constraints.

The first key server we discuss is of the *friendly backup* type where each server is trusted by the clients in the system, but they each serve a different group of clients – this could be because of locality, efficiency etc.

The second implementation we discuss is based on the *mutual distrust* environment where each server acts as a trusted primary for a given group of clients, but the client group does not completely trust any of the other servers, although they are willing to combine an aggregation of trust between a group of backups to provide continued service, albeit at a reduction in performance.

Several trusted Primaries

The scenario we envisage here is where a number of key servers provide service to a large community of users. Implicit in our assumptions is that all clients trust all key servers available to them.

We build our example around the Needham and Schroeder [NS78] protocol, which requires that each of the key servers has a communication key with each group member. If the number of users in the group is n and the number of servers is m this leaves us with $m \times n$ keys in the system. This is not envisaged to be a problem, as $n \gg m$, which leaves us with much fewer keys that in the n^2 worst case.

To outline: There are a group of clients C_i (where $i = 1 \cdots n$) and a group of servers S_j (where $j = 1 \cdots m$). Within the client community there are m groups G_j who have S_j acting as their Primary. Each client has stored locally a list B_i, which is a list of the servers which the client uses as a backup in case their primary is unavailable. The permutation of servers in each clients list will be diverse throughout each group G_j in order not to flood a single server when S_j is unavailable.

Figure 7.1 shows the message interactions between a client C_1 and its primary S_1 and a client and its first backup S_2 where the client is trying to obtain a session key to communicate with a second client C_2.

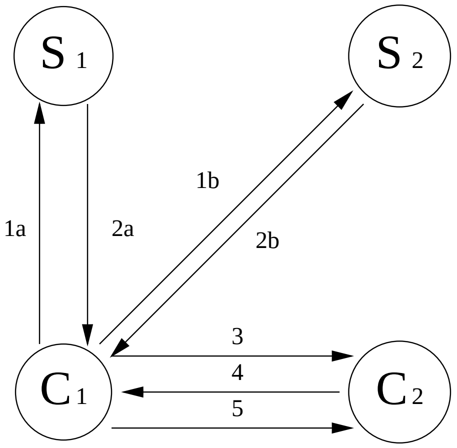

Figure 7.1: Trusted Primary and Trusted Backups

The content of each of the messages are as follows:

$(1a)C_1 \rightarrow S_1 : C_1, C_2, N_{as_1}$
$(2a)S_1 \rightarrow C_1 : \{N_{as_1}, C_2, K_{c_1c_2}, \{K_{c_1c_2}, C_1\}_{K_{c_2s_1}}\}_{K_{c_1s_1}}$

$(1b)C_1 \rightarrow S_2 : C_1, C_2, N_{as_2}$
$(2b)S_2 \rightarrow C_1 : \{N_{as_2}, C_2, K_{c_1c_2}, \{K_{c_1c_2}, C_1\}_{K_{c_2s_2}}\}_{K_{c_1s_2}}$

$(3)C_1 \rightarrow C_2 : \{K_{c_1c_2}, C_1\}_{K_{c_1s_1}}$ **or** $\{K_{c_1c_2}, C_1\}_{K_{c_1s_2}}$
$(4)C_2 \rightarrow C_1 : \{N_{c_2}\}_{K_{c_1c_2}}$
$(5)C_1 \rightarrow C_2 : \{N_{c_2} - 1\}_{K_{c_1c_2}}$

When C_1 wishes to communicate with C_2 he first of all sends message $1a$ to his primary (in this case S_1), if he does not receive message $2a$ within a pre-set time limit, he assumes that S_1 is unavailable (due to either server crash or communication failure). He then goes ahead and sends message $1b$ to S_2. In this example we will assume that the first backup is available, and that he receives message $2b$ within the timeout period. The protocol then continues as normal and both clients can communicate with $K_{c_1c_2}$.

If message $2a$ is just delayed past the timeout period, C_1 will then discard message $2a$ upon its arrival.

We note that we initially decided to try and use the wide-mouth-frog protocol [BAN89] in our example. The attraction of that particular protocol was the simplicity of the protocol structure.

Unfortunately we found that this protocol was not suitable. The reason being that, if C_1 does not get an immediate response from S_1 as part of the protocol, then he does not know when to contact S_2 as part of the backup procedure. Having the protocol fail in this manner also puts some ambiguity into the fault diagnosis, as C_1 cannot be sure if it is S_1 or C_2 which is currently unavailable.

Single trusted Primary

In this example we have a scenario where a client C_i is willing to trust its primary S_j, but is not willing to trust a single individual server in its list of backups B_i.

We illustrate below with a protocol where C_1 is requesting C_2's public key.

Figure 7.2 shows the different runs of the protocol on the same diagram. Messages $1a$ and $1b$ show the normal execution of the protocol if C_1 is able to contact his primary. We highlight two options for a backup protocol. The first is shown by messages $1b$, $2b$, and $3b$, with the second shown by messages $1c$, $2c$, $3c$ and $4c$. Both protocols use S_2 and S_3 together to increase C_1's confidence in the result.

We omit the final part of the protocol interaction with the client C_2 for brevity.

All encryption is done via public key, with $\{M\}_{K_P^{-1}}$ used to show the encryption of message M under P's private key, and $\{M\}_{K_P}$ used to show encryption of message M under P's public key.

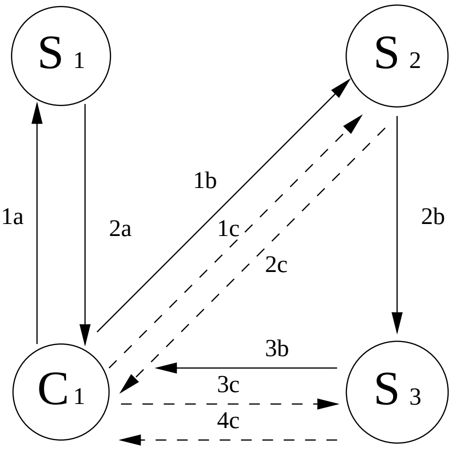

Figure 7.2: Trusted Primary and untrusted Backups

The message contents are as follows:

$(1a)C_1 \rightarrow S_1 : \{\{C_1, C_2, N_{c1}^1\}_{K_{c_1}^{-1}}\}_{K_{S_1}}$
$(2a)S_1 \rightarrow C_1 : \{\{C_2, K_{C_2}, N_{c1}^1\}_{K_{s_1}^{-1}}\}_{K_{c_1}}$

$(1b)C_1 \rightarrow S_2 : \{\{C_1, C_2, N_{c1}^2\}_{K_{c_1}^{-1}}\}_{K_{s_2}}, \{\{C_1, C_2, N_{c1}^3\}_{K_{c_1}^{-1}}\}_{K_{s_3}}$
$(2b)S_2 \rightarrow S_3 : \{\{C_2, K_{c_2}, N_{c1}^2\}_{K_{s_2}^{-1}}\}_{K_{c_1}}, \{\{C_1, C_2, N_{c1}^3\}_{K_{c_1}^{-1}}\}_{K_{S_3}}$
$(3b)S_3 \rightarrow C_1 : \{\{C_2, K_{c_2}, N_{c1}^2\}_{K_{s_2}^{-1}}\}_{K_{c_1}}, \{\{C_2, K_{c_2}, N_{c1}^3\}_{K_{s_2}^{-1}}\}_{K_{c_1}}$

$(1c)C_1 \rightarrow S_2 : \{\{C_1, C_2, N_{c1}^4\}_{K_{c_1}^{-1}}\}_{K_{s_2}}$
$(2c)S_2 \rightarrow C_1 : \{\{C_2, K_{C_2}, N_{c1}^4\}_{K_{s_2}^{-1}}\}_{K_{c_1}}$
$(3c)C_1 \rightarrow S_3 : \{\{C_1, C_2, N_{c1}^5\}_{K_{c_1}^{-1}}\}_{K_{s_3}}$
$(4c)S_3 \rightarrow C_1 : \{\{C_2, K_{C_2}, N_{c1}^5\}_{K_{s_3}^{-1}}\}_{K_{c_1}}$

C_1 begins by sending message $1a$ to his primary S_1. If he does not receive message $2a$ within a timeout period, then he starts the section of the protocol involving the backup servers. In the case where the message arrives after the timeout period, C_1 will use the key received in message $2a$ to communicate with C_2, and does not use the key received by the backup protocol. We use this mechanism in preference to the previous method of dropping the reply from the primary because C_1 trusts S_1 explicitly.

If C_1 can be sure that servers S_2 and S_3 are alive and are going to talk to each other and himself, then he can use the the protocol mechanism that delivers messages $1b$, $2b$ and $3b$ in round robin fashion. This protocol requires that C_1 trusts S_2 and S_3 enough to respond and co-operate enough with him, but not enough to provide him with a valid public key[2]. As this is an unlikely scenario, it is more likely that he uses the protocol mechanism that uses messages $1c$, $2c$, $3c$ and $4c$. This is a more expensive protocol, but if either backup is unavailable, then it means less overhead, as C_1 can send out messages to other servers on his backup list until he receives enough replies to aggregate the trust to his satisfaction.

We have shown here a protocol that utilises only two servers as backups, but in a real scenario, C_1 would require a larger number in order to justify which version of the public key he uses[3]. Explicitly, if the number of backups he relies upon is odd, then it becomes a simple N-model redundancy method, with an output only being accepted if greater than $N/2$ keys match.

7.6.2 Notarization

The type of server implementation we discuss here is one where the threat is denial of service.

Notarization is discussed in the literature by Schneier and Kelsey [SK96] and Haber and Stornetta [HS91].

In our example of notarization using a primary backup based method, we sketch out a high level design for the computerisation of the patent application mechanism.

It is an existing real world example of a service that – amongst other things – needs to provide a reliable and secure notarization mechanism. Some of the requirements put forward in the paper by Haber and Stor-

[2] We note the connection between this work and the discussion of protocol suitability with regard to timing considerations from section 6.5.2.

[3] If the protocol uses just two backups and their replies do not match, then C_1 is unable to chose between returned values of the key. There is an exception to this rule, where the reason C_1 does not trust either backup explicitly is that either might not have C_2's current private key, and compensating by using the key with the most up to date certificate.

netta [HS91] do not have to be met when creating a notarization service for the Patent Office. There is no need for long term secrecy[4] (unless the government has some separate method for patenting devices not in the public domain for military purposes), as clients registering patents will not concerned about the secrecy of the documents registered with the timestamping service. This requirement is not necessary with the example of a Patent Office. Also, their method of using hashes to overcome the storage problem does not work, given that the Patent Office needs to hold a copy of all the patent applications they receive.

There are other benefits to the Patent Office from having their system online. Firstly, their methods of checking back through previous patents can be improved with computerised information retrieval methods. Secondly, under the system currently in place, they only provide stamps which authenticate on which day the patent application was received, they would be able to use a hash chaining method to demonstrate in which order all patent applications arrived, achieving a finer degree of granularity.

There is good reason for the Patent Office to require its registration application to be highly available, as they do not want any mad inventors being upset at not being able to file the patent for their latest *helpful-widget*, lest they get restless, and turn their minds to developing an *anarchy-widget*.

Our assumption in this scenario is as follows: The Patent Office is in control of all the servers, and each server is implicitly trusted to provide a current date stamp, and issue a valid certificate of receipt.

There are a variety of solutions to selecting the primary in the Primary-backup approach. The differences are mainly focused around how they solve the partitioning problem, Davidson et al. [DGMS85] give a discussion about the various solutions in the literature. We will restrict ourselves to a solution that only ever provides one primary. We do this in order to simplify the mechanism used to generate the hash chain. The problem with allowing the service to run more than one primary is the reconciliation of state across the multiple primaries[5].

To facilitate this solution, we require that in case of partitioning, greater than half of the servers can form a majority (i.e., given N servers, at least $(N/2) + 1$ servers can form a group). Once these recognise the fact that they can form a group, they elect a new primary from their number. Any other live servers that find themselves in a group which is not a majority

[4] Although the initial patent application should be secret to avoid an evesdropper gaining access to the patent information before the request has been confirmed by the server.

[5] Reconciliation of hash chains across multiple primaries after partitions are reconnected is in itself a difficult task without a highly accurate time service.

wait until the fault is corrected and then join the majority partition.

When M (the resident mad inventor) decides to submit a patent for his latest *helpful-widget*, he is required to send in the following information:

- Personal details - This will generally be in some list format, containing such information as full name, address, etc.
- Patent - This will be an electronic copy of the full patent information.
- Hash of Patent - An electronic hash of the patent information. This will be used by the service in the generation of the receipt.

The client then receives a receipt for his patent application from the server. This will contain the following information:

- Personal details - In order to tie the receipt to the individual.
- PID (Patent Identification information) - A unique identifier that the patent office uses internally to track applications, and the client uses in future consultation.
- Hash of the Patent - A copy of the hash of the patent details.
- Date - The current patent issuing mechanism only gives granularity of one day, thus we will imitate this here. Further control over granularity is given in our system by use of linking the hashes together. This removes the need for the patent service to have a reliable clock.
- Link information - A hash of some information from the previous patent. This provides us with a mechanism to show which receipts were given in which order.

The simplified protocol is shown below with M to denote the client applying for the application, and S to identify the server. C_m is M's public key certificate[6], $M.I.D.$ are the personal details, with P and $h(P)$ denoting the patent and a hash of it respectively. L_{n-1} is the link information to the previous patent application. We outline below the content of this link information, where $h(G_{n-1})$ is a global running hash updated with each new patent application:

[6] It is not at first obvious that we need a full certificate of the public key here – who would want to file a patent in someone else's name? – but it does prevent some mad inventor submitting a patent under a competitors name that has the men in dark suits knocking on the competitor's door.

$$L_{n-1} = P.I.D._{n-1}, h(P)_{n-1}, h(G_{n-1})$$

$M \rightarrow S : \{C_m, \{M.I.D., P, h(P)\}_{K_m^{-1}}\}_{K_s}$
$S \rightarrow M : \{\{P.I.D., h(P), L_{n-1}, h(L_{n-1}), D\}_{K_s^{-1}}\}_{K_a}$

Server details

In the above protocol we make the assumption of a single server, we will now elaborate on the replication mechanism and how it affects both the client-server communication, and each server's handling of new applications.

In order to contact the patent office, the client picks one of the servers at random[7] (their addresses, along with current public key information, will of course be published in a reliable public source such as "Mad Inventor Monthly"). If the server is currently down or unreachable, then the client will wait for a timeout period before trying one of the other servers.

Upon making contact with a live server, the client will receive one of three replies depending on the situation of the server. Any live server will be in one of three states. It will either be the current primary, a backup in the primary partition, or a backup in a minority partition. The response will be as follows:

Primary Processes the request as normal, replying to the client upon completion.

Backup (primary partition) Forwards the request to the primary for processing, notifying the client that it has done so. This is to ensure that if the primary is busy, and its response to the request delayed, then the client does not assume that the backup he contacted is unavailable.

Backup (minority partition) Replies to the client, informing him that as it is not in the same partition as the primary it is unable to deal with the request further. It should also send back a list of those replicas which are in the same partition as it, this will stop the client from trying these unnecessarily.

How an update of the patent application state is carried out depends on the method by which the information sharing is done between server replicas.

[7] In the literature, it is noted that the client should know which is the current primary, this is feasible in a small environment such as a LAN, but in a large open environment such as we envisage for this scenario, it would be difficult to keep all possible clients informed as to which server is the current primary.

We identify two possible means of maintaining state within the system. The first method is where the servers each have access to a shared disk mechanism, and the current primary is the only one with a token to access the disk. Any writes to the disk will have to be done atomically. The disk itself should also have some backup mechanism, although it need not be as extensive as the server replication mechanism. The second method is where each client stores their own copy of the current system state. This would add overhead to the system, which could get unwieldy if a large number of patent applications are made. We also envisage a compromise solution, where the information needed to service a patent application request (copy of the global hash etc.) is stored locally with each server, and the actual patent content separated onto a global disk, with both sets of information being reconciled when it comes to the task of manually checking the validity of the patent's claims. This task is, in effect, done off-line from the patent application processing services.

Change of topology

Assuming a normal running scenario, we have a majority partition, with one server acting as primary, then a number of things can happened from here.

Firstly, one of the backups can return into the group (if there was a prior partition). This does not affect the running of the service, but it is the task of the current primary to bring the new backup up to date with the current system state.

Secondly, one of the backups within the partition can be lost from the partition. This will either have no effect on the service (i.e., there is still a majority of servers in the primary's partition) or it will reduce the number of servers within the current primary partition to below a majority. In this case the primary will have to suspend acting as a primary, and the replicas will have to wait until a majority partition is re-formed.

The third thing that can happen is that the primary can crash. If the new partition is now below a majority, then the system has to wait until the partition returns to a majority. If the new partition still holds a majority of servers then there needs to be an election of a new primary. To make the election easier, there is a number associated with each server, it is the server with the highest number within the partition that takes on the role of the primary.

If we assume that the processing of a patent application takes a relatively short period of time in comparison to the expected inter-arrival time of new applications, then we can assume that if the primary crashes, then there are either none or one applications that have yet to be processed to completion.

If the primary crashes with a queue of unprocessed requests, then unless each of the backups has a copy of the information required, it is impossible to ascertain in which order the unprocessed requests should be dealt with. To solve this, we can introduce a broadcast mechanism within the group that currently hosts the primary. When a group member receives a request it forwards it to the primary which then broadcasts it to the rest of the group, such that each backup has knowledge of which order to process outstanding requests. Care should be taken with the broadcast protocol, in that the primary should sign the broadcasts, as the sequence should be binding between each instance of the primary.

The complete system we envisage would operate in the following manner (assuming the information is arriving at a live server in the same partition as the primary):

1. Patent information arrives at the server.
2. The server logs the main patent details to a shared media such as a disk, with its own backup facility.
3. The server forwards the details to the primary, who securely re-broadcasts the order in which it receives them.
4. The primary processes the requests in the queue, reconciling the patent application information with the patent itself on the shared media.
5. The primary broadcasts to the other members of the group when it finishes the request[8].

For added confidence in the system, the list of signed n-tuples containing the patent application number, the hash of the patent information and the global running hash could be published in a public media (e.g. “The Times” or “Mad Inventor Monthly”).

Because of the on-line nature of such a mechanism, it means people will be able to submit patents without having to travel to London if they do not wish to incur the delay of postage, and it could also be a 24-hour system, which would be an advantage, as all the best mad inventors do their best work at night.

[8] Note that messages 3 and 5 can be piggy-backed for efficiency.

7.6.3 Recovery Cache

We outline a simple idea of using the primary-backup mechanism as an off-site secure recovery cache.

A server of this type would be able to provide a means of securing an off-site backup in case of a dire emergency, possibly as part of some disaster recovery mechanism.

It would be a simple mechanism where a client contacts the current primary, and the primary then co-ordinates the update of the recovery information between the current backups.

When registering, the client is given information that allows him to retrieve the necessary recovery information at a future date when he needs it. This would include an identifier token with an access capability, this would then make sure that only a valid client would be able access the data. Another piece of information would be a hash of the stored information. This would allow the client to check that the information retrieved was the same as the information deposited. If the service was being paid for, the hash would need to be signed in order that a client could pursue further damages against the service in the case where the service was throwing away all data received, and then returning random garbage in the case of a request.

Although we only briefly describe such a service here, it would be an interesting area to research further. Particularly which type of replication primitives would be best suited for such a task.

7.7 Conclusions

We look at the Primary-backup method of replicating servers as a method of providing resilience in security servers. We have modified the standard Primary-backup ideas in the literature for our own methods, outlining various changes to the general theme.

We then provide some examples of using Primary-backup in scenarios where we believe our variations can help replicate servers in a reasonably efficient manner.

We envisage developing this theme for providing servers and other security mechanisms that can survive denial of service attacks in a lightweight form of replication. We believe this opens up a promising avenue of research which deserves to be explored more fully within the research community.

A further line of research in this area would be to see which qualities we need to change in our design when it comes to building these types of services. In our example on the key server, we see that protocols such as wide-mouth-frog are not very suitable for such mechanisms. It would be interesting to discover what other properties are constrained by such replication.

Chapter 8

A distributed object server

8.1 Overview

In this chapter we overview the design of a service for distributing an object among many servers to provide for resilient access to the object. Our design centres on a concept of providing the object owner with the ability to regulate the object management. The backbone to this design uses the State Machine approach of replication. Our design differs from current designs that adopt the approach to a byzantine environment, by allowing the client to control the group management sections of the protocols.

8.2 Introduction

Our method uses an extension of some existing work in the area, with the main modification being directed by our notion that the control of the fault tolerance within the design can be regulated by the client.

Existing work in the area is predominantly that done by Reiter on *Rampart* [Rei96a, Rei94b, Rei94a]. His work is a natural extension of the work on *Isis*[BSS91, RBG92] and is built around a set of group broadcast primitives. These group broadcast facilities are based on view mechanisms that are administered within the group.

The backbone of our work in this design is based upon our advances in this direction, and takes up the majority of the remainder of this chapter. The other sections of this chapter are concerned with the object management, group structure management, and access control structures respectively. We start off with a brief description of our computational model, and then address each of these areas in turn.

8.3 Computational model

Our computational model is based on an Internet style environment, where there is a point-to-point asynchronous communication channel. Our interest in this type of environment stems from the need for a client to have a reliable distributed store external to their own domain. There is a similarity between this and the motivation behind the *Eternity server* [And96], but our work differs in the update properties provided, and the fact that the eternity server provides for anonymity. The eternity server requires anonymous posting, and an ability to post to the service, without the ability to delete. A similar service was specified by Blaze [Bla96], but his system has a different set of update properties.

We assume a global set of servers $\mathcal{S}$ that are accessible to a global set of clients $\mathcal{C}$. Each client c_m wishing to use the service can then specify a subset S_{c_m} of these servers to use in order to store an object (as shown in figure 8.1).

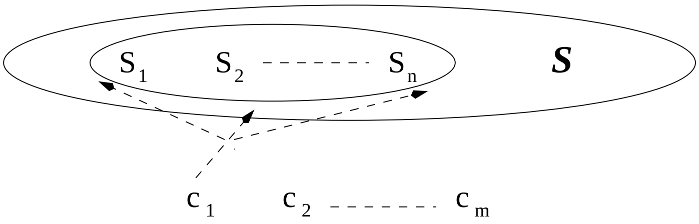

Figure 8.1: System model: Client and server groups

We assume that a pre-defined number of servers in each group allocated by a client will remain *correct* and that any others are allowed to be *byzantine faulty* (i.e., exhibit arbitrary malicious behaviour). We assume that within a group of servers selected by a client to act as a repository, there will be at most $\lfloor \frac{n-1}{3} \rfloor$ faulty servers. This limit is tight within asynchronous distributed systems, and an elegant proof is provided by Bracha and Toueg [BT85].

We also assume that there is some public means by which a site can securely notify their intent to provide service in this environment (e.g., a globally accessible newsgroup).

8.4 Broadcast protocols

In this section we describe the underlying communication mechanism that the service is built upon.

In the environment we are designing for, achieving agreement is known to be a difficult task. Fischer et al. [FLP85] demonstrate that even in an asynchronous environment limited to *crash failures* (i.e., failures that are benign in nature, and result in the loss of a process), it is impossible to deterministically achieve consensus [1], even in the event of a single unannounced process death.

Circumventing this impossibility result is done in one of two ways. Initially, this was achieved using randomised protocols [CD89, BT83, BO83, BT85]. These work by allowing a possible non-termination at each round, but demonstrate that if round iteration carries on for a period of time, then termination is reached with probability 1. The second mechanism used to circumvent the result is based on the notion of *unreliable failure detectors* [CT96, MR97c] and demonstrates that if failure detectors can meet certain realistic goals, then termination is possible.

Atomic broadcast is a related problem to consensus. One notable system that has solved the problem of atomic broadcast in an asynchronous byzantine system is the work done by Reiter on Rampart [Rei96a, Rei94b, Rei94a]. He achieves this by using membership changes in the group to allow the atomic broadcast protocol to continue. Group membership was conjectured to be possible in a totally byzantine environment, but since the completion of this work, Chandra et al. [CHTCB95] demonstrate that group membership itself is an impossible task in an asynchronous environment[2].

Our method of achieving atomic broadcast is related to the work carried out by Malkhi and Reiter [MR97c] on using intrusion detection for byzantine consensus. We also use the randomness inherent in a network, defined by Bracha and Toueg [BT85] as a *fair scheduler*, which allows a process group to achieve consensus.

8.4.1 Reliable Broadcast

As a building block for our atomic broadcast protocol, we have a reliable broadcast protocol. This work is a simple extension of the work done by others [HT94, Rei94b]. A reliable broadcast protocol needs to satisfy the following properties. In the definitions, p, q and r represent processes which are group members. Also m and m' represent messages sent between processes. *bcast-send* and *bcast-receive* are the primitives provided by the protocol. In

[1] We use consensus to mean an agreement reached by all correct processes, and not a simple majority.

[2] Although they do note that is should be possible circumvent their impossibility result using similar methods to those employed to circumvent the impossibility of consensus.

these definitions and in the subsequent algorithm, we assume the existence of underlying point-to-point mechanisms *send* and *receive*.

Integrity: For all p and m, a correct process executes a *bcast-receive*(m, q) at most once and, if q is correct, only if q executed *bcast-send*(m).

Agreement: If p and q are correct and p executes *bcast-receive*(m, r), then q executes *bcast-receive*(m, r).

Validity: If p and q are correct and p executes *bcast-send*(m), then q executes *bcast-receive*(m, p).

Source Order: If p and q are correct and both execute *bcast-receive*(m, r) and *bcast-receive*(m', r), then they do so in the same relative order and, if r is correct, in the order in which r executed *bcast-send*(m) and *bcast-send*(m').

In their work on consensus in distributed systems, Malkhi and Reiter [MR97c] also require their underlying broadcast protocol to provide Causal Order. We exclude this criteria from our reliable broadcast primitive for three reasons:

1. In their work on a modular approach to building broadcast primitives, Hadzilacos and Toueg [HT94] demonstrate that Causality is not necessary to achieve Total Order, and dropping the requirement makes the protocol more lightweight.

2. Reiter and Gong [RG95] demonstrate that true Causality in a byzantine environment may be unachievable.

3. Given our approach to solving total order has a leader propose the next request to process based on external actions of the client, using a causal mechanism within the group broadcast would be meaningless without extending it to the client base, and this would increase the complexity greatly[3].

The algorithm for reliable broadcast is shown in figure 8.2, where all messages are signed by the originator. The different message contents and their meanings are summarised in table 8.1.

A process immediately receives any send which it makes as part of a broadcast, and counts its own vote in the algorithm for *bcast-send*.

[3] In a loosely coupled system, this requirement could prove impossible.

$<B1,\ l,\ p_i,\ m>$	p_i sending m as l'th message
$<B2,\ l,\ p_j,\ p_i, m>$	p_j agreeing to having seen m as l'th message from p_i

Table 8.1: Message meanings in the reliable broadcast algorithm

```
pseudocode for bcast-send(m)

  send $<B1,\ l,\ p_i,\ m>$ to all $p_j$ $(j \neq i)$
  wait to receive $\lceil \frac{2n+1}{3} \rceil$ $<B2,\ l,\ p_j,\ p_i, m>$
    forward $<B2,\ l,\ p_j,\ p_i, m>$ to all $p_j$
    bcast-receive($m$, $p_i$)

pseudocode for bcast-receive($m$,$p_j$)

  if receive $<B1,\ l,\ p_i,\ m>$
    send $<B2,\ l,\ p_j,\ p_i, m>$ to $p_j$
  if receive $\lceil \frac{2n+1}{3} \rceil$ $<B2,\ l,\ p_j,\ p_i, m>$
    bcast-receive($m$, $p_j$)
```

Figure 8.2: *bcast-send* and *bcast-receive* at process p_i

8.4.2 Hybrid Atomic Broadcast

We use a hybrid approach to solve the atomic broadcast problem. We require a hybrid approach in order to give our protocol two properties required in an asynchronous distributed environment.

Liveness This property is required to ensure that the protocol can progress in the face of unforeseen failure to a group member.

Termination This property is required to ensure that the protocol can complete and that it does not enter an infinite run[4].

To fulfill the first property of liveness, we employ the use of *weak failure detectors* [CT96]. We only require that our failure detector satisfies the

[4] We highlight the difference between this and *Terminating Reliable Broadcast* [HT94] where the broadcast should terminate even if the client initiating the broadcast does not send the message, as this is impossible in an environment where the client requests are unpredictable.

criteria of *Strong Completeness* – that eventually every process that does not send out a broadcast is permanently suspected by every correct process.

To fulfill the second property of termination, we rely upon the randomisation inherent in the system, which is the design principles behind Bracha and Toueg's algorithm to provide consensus [BT85]. Their principle of a *fair scheduler* states that, there is a constant probability that all processes receive messages from the same set of correct processes. We believe that such a property is possible in such a large scale distributed environment as envisaged in our design.

Another choice we could have made in order to ensure termination would be to accept the full specification of an eventually strong failure detector ($\diamond\mathcal{S}(bz)$) [5] for a byzantine environment, which conforms to the added property of *Eventual Weak Accuracy* – which states that there is a time after which some correct process is never suspected by any correct process.

We believe an implementation of an eventually strong failure detector in our environment to be a non-trivial task. For all correct processes in a group to eventually reach a state of not suspecting the same correct process implies that the failure detector shares state between correct processes[6]. We believe this to be a drawback as it implies one of two things. Either the failure detectors share state within the group and have regular updates, which goes against our notion of a use-only group structure, or there is some global state between all possible servers, where an update protocol which provides the completeness required for total order would be prohibitively expensive.

A third choice we could have made would be to use randomness in the protocol itself, which implies access to a good source of random numbers. Methods of achieving consensus using random numbers have tended to use a trusted distributor for a random coin [Per84, Rab83] – something we do not want rely upon, or an underlying byzantine agreement algorithm to decide the outcome of a global coin toss [FM85, BO85] – which would prove prohibitively expensive [7].

In addition to the properties provided by the reliable broadcast protocol and those dictated for liveness and termination, the atomic delivery protocol has to provide total order [HT94] as defined below [8]:

[5] As defined for a byzantine environment by Malkhi and Reiter [MR97c].

[6] Conversely, *Strong Completeness* is relatively easy to achieve using local time-outs and no accumulation of global state.

[7] Given that the coin toss needs to be both random (itself relatively easy to achieve) and unanimous amongst all correct servers.

[8] We assume the existence of mechanism *a-deliver*, which implements atomic delivery at all correct processes.

Total order If correct processes p and q both *a-deliver* messages m and m', then p *a-delivers* m before m' if and only if q *a-delivers* m before m'.

Client to server communication

When a client wishes to issue a request to the service, it forms a request token, signs it, and forwards it to each member of the group. A malicious client may send the token only to a subset of servers, this does not cause a problem, any such discrepancies are dealt with in the inter-server section of the broadcast protocol. The structure of the request token is shown below:

$$<O.I.D.,\ C.I.D.,\ N_c,\ T_o>$$

The O.I.D. represents the object identifier, the C.I.D. is the client identifier for the client issuing the request, N_c is a sequence number, and T_o is the token that will be described in section 8.6.2.

The first two of these are easily verified by the service, and any request that does not contain a valid pair is automatically rejected. The sequence number is used only to ensure the freshness of the request in that it is not a replay, and cannot be used be used by the service to asses the timeliness of the request.

From here on we will talk in terms of the request identifier (R.I.D), which is a concatenation of the object identifier, the client identifier and the sequence number. The R.I.D. uniquely identifies a valid request.

When the servers enter the server-to-server section of the communication, they check to see if the request identifier has been used before. If a malicious client issues a different T_o with the same concurrent request identifier, a server will discover the replay and then notify the other servers in the group, and the request will automatically be denied.

Because a malicious server cannot falsify a request identifier, no threshold is required to constitute proof of malicious action from a client.

Server to server communication

The server to server communication which provides the group with atomic broadcast is based on a round robin based algorithm. This is an extension of the work done by Malkhi and Reiter [MR97c], which is itself an extension of the paxos algorithm [Lam89].

The basis of the algorithm is a round based mechanism, with a *leader* in each round responsible for suggesting the next message to be delivered in the total order queue. The leader is rotated around the group in a round robin fashion, in the order that the servers appear in the server list used by the object owner when generating the object[9].

Each server maintains a local queue, which holds valid requests received from clients, as well as requests forwarded from other servers. Once delivered by the atomic broadcast protocol, requests are taken from this queue to be processed on the object. Figure 8.3 has a diagram of the input queue used locally at each server to allocate incoming requests. When it is a correct group member p_n's turn as *leader* it takes the next item off the head of this queue and suggests it as the next transaction to process in total order.

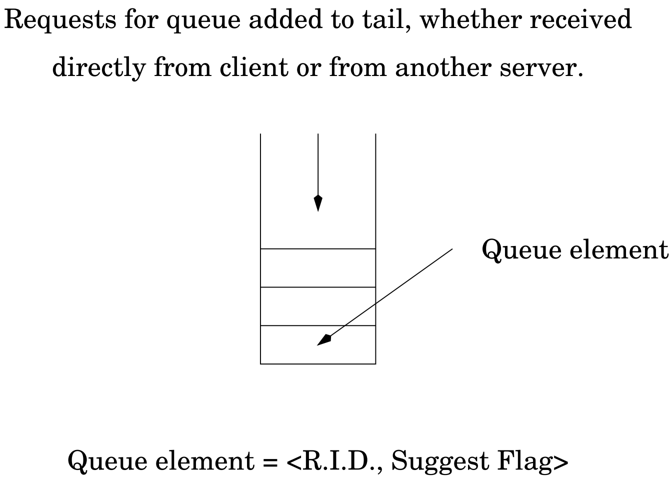
Requests for queue added to tail, whether received
directly from client or from another server.
Queue element
Queue element = <R.I.D., Suggest Flag>

Figure 8.3: Overview of queue held at server for total order

The information stored in the queue is listed as following:

R.I.D. The request identifier.

Suggest Flag This flag is set on the request currently proposed for transferal to the total order processing queue by the leader for the current round.

When a server receives a request from a client (or when it receives a valid request which it has yet to receive directly from the client from another group member as a suggestion), then it places it on the tail of the queue. Once a request is accepted, then it is taken from the incoming queue, and processed by the server.

The algorithm for atomic broadcast is shown in figure 8.4, where all messages are signed by the originator. The different message contents and

[9] Details of the formation of new groups is discussed later on in section 8.5.

their meanings are summarised in table 8.2. Only the leader of round r can form a valid proposal or rejection.

$<A1,\ p_i,\ r,\ R.I.D.>$	Propose R.I.D for total order
$<A2,\ p_i,\ r,\ R.I.D.,\ ACK>$	Accept R.I.D. for total order
$<A3,\ p_i,\ r,\ R.I.D.,\ NACK>$	Reject R.I.D. for total order
$<A4,\ p_i,\ r,\ R.I.D.>$	Suspect leader for round r
$<A5,\ p_i,\ r,\ R.I.D.>$	Suggest to add R.I.D. to queue and start next round

Table 8.2: Message meanings in the total order algorithm

If any correct process receives a message from any other process that they suspect the leader for a round r, and they also receive the evidence from the leader for round r that is sufficient to accept the request into total order, then they forward this evidence they received from the leader of round r to the process that notified their suspicion.

If we regard the l'th request processed at a process p as p_l and the *leader* for that round is denoted $leader_l$, a proof of correctness for the algorithm presented in figure 8.4 is outlined below. Theorem 8.1 demonstrates that the algorithm provides the group with total order, and theorem 8.2 demonstrates that the properties of liveness and termination are preserved.

Theorem 8.1 *Any two correct processes p and q, that process $R.I.D._1$ and $R.I.D._2$, process $R.I.D._1$ and $R.I.D._2$ in the same order.*

Proof. (Sketch) Suppose that p processes $R.I.D._1$ as request p_l and request $R.I.D._2$ as request p_{l+1}, then $leader_l$ must have forwarded $\lceil\frac{2n+1}{3}\rceil$ $<A2,\ p_i,\ l,\ R.I.D._1,\ ACK>$, and $leader_{l+1}$ must have forwarded $\lceil\frac{2n+1}{3}\rceil$ $<A2,\ p_i,\ l+1,\ R.I.D._2,\ ACK>$. Thus q must have received the same proofs and also processed the requests in their respective rounds. □

Lemma 8.1 *For any round r in which all correct processes participate and the leader is correct, then the leader can advance the total queue.*

Proof. (Sketch) If a correct leader in a round can contact all processes in a round, then $leader_r$ receives $<A2,\ p_i,\ r,\ R.I.D.,\ ACK>$ from each of all correct processes, and can subsequently generate the necessary evidence to promote the current $R.I.D.$ from the input queue to the processing queue. □

```
if $r$ mod $n = i$ ($p_i$ is the leader) and queue not empty
  send $<A1,\ p_i,\ r,\ R.I.D.>$ to $p_j$ ($\forall j \neq i$)
  wait for $<A2,\ p_j,\ r,\ R.I.D.,\ ACK>$ from $\lceil \frac{2n+1}{3} \rceil$
    forward $<A2,\ p_j,\ r,\ R.I.D.,\ ACK>$ to all $p_j$
    accept R.I.D. and enable round $r + 1$
  or suspect $\lfloor \frac{n+1}{3} \rfloor$
    send $<A3,\ p_i,\ r,\ R.I.D.,\ NACK>$ to all $p_j$
    reject R.I.D. and enable round $r + 1$

if $r$ mod $n \neq i$ ($p_i$ not the leader)
  wait ($<A1,\ p_l,\ r,\ R.I.D.>$ from leader)
    or ((queue not empty) and (suspect leader))
  if leader proposes $<A1,\ p_j,\ r,\ R.I.D.>$
    send $<A2,\ p_i,\ r,\ R.I.D.,\ ACK>$
    if leader sends $\lceil \frac{2n+1}{3} \rceil$ $<A2,\ p_j,\ r,\ R.I.D.>$
      accept R.I.D. and enable round $r + 1$
    if leader sends $<A3,\ p_l,\ r,\ R.I.D.,\ NACK>$
      reject R.I.D. and enable round $r + 1$
    if not reply from leader and suspect leader
      send $<A4,\ p_i,\ r,\ R.I.D.>$ to all $p_j$
        wait for $<Mx,\ p_i,\ r,\ R.I.D.>$ or suspect all
          if receive $\lceil \frac{2n+1}{3} \rceil$ $<A2,\ p_j,\ r,\ R.I.D.,\ ACK>$
            accept R.I.D. and enable round $r + 1$
          if receive $<A2,\ p_l,\ r,\ R.I.D.,\ NACK>$
            reject R.I.D. and enable round $r + 1$
          if suspect all
          bcast-send $<A5,\ p_i,\ r,\ R.I.D.>$
  if queue not empty and suspect leader
      bcast-send $<A5,\ p_i,\ r,\ R.I.D.>$
  if bcast-receive $\lfloor \frac{n+1}{3} \rfloor + 1$ $<A5,\ p_j,\ r,\ R.I.D.>$
      place R.I.D. queue and enable round $r + 1$
```

Figure 8.4: Atomic broadcast algorithm at p_i

Lemma 8.2 *There exists a round r after time t, at which a correct leader can contact all correct processes.*

Proof. (Sketch) If the network behaves according to a *fair scheduler* [BT85], then there is a positive probability that at any round, any given correct processor does not suspect any other correct processor within that round. Subsequently, there exists (with probability 1) a round, after time t in which all correct processors do not suspect any other correct processors. □

Theorem 8.2 *Eventually all requests from correct clients are processed.*

Proof. (Sketch) If a correct client sends a request $R.I.D._c$ to the service, then a correct instance of the server (p_r) will receive the request and place it in their input queue. As the input queue of p_r is FIFO in that p_r will always propose the head of the queue, and the liveness of the queue is guaranteed from Lemma 8.1 and Lemma 8.2, then $R.I.D._c$ will eventually be processed. □

8.5 Group membership

Control of the group membership is handled by the owner of the object in our system. Changes in group membership only happen when the object owner wishes to change the group membership.

In previous work that uses group oriented systems [BSS91, Rei96b] they call an instance of a group membership a *view*. Changes to a group view are needed in order to overcome the impossibility result of Fischer et al. [FLP85], but because we use a different method to overcome this result we need not force view changes when a member of the group is unreachable.

8.5.1 Group initialisation

The initialisation of the group membership is carried out by the owner [10] of the object. An owner who wishes to deposit an object with this service consults the globally accessible directory to ascertain which servers are available to store information.

The object owner then decides which of these information servers it is willing to use in the creation of a group.

[10] The owner of an object is one of the set of clients in the system.

Bracha and Toueg [BT85] highlight that a distributed agreement protocol should not start unless there is a threshold of servers that are party to the protocol. Our protocol circumvents this problem in two ways:

1. The atomic broadcast protocol starts by having the group leader for the initial round forward the group creation request to all members on the server list. This will ensure two things:
    (a) The number of group members that see the request rises above the given threshold, thus allowing the agreement process to begin.
    (b) It allows the proposed group members to check whether they are party to a previous / concurrent request from the same client to set up the same object identifier[11].
2. Unless a threshold of servers ($\lceil \frac{2n+1}{3} \rceil$) agree on the group composition, then the atomic broadcast protocol does not progress past its first round, and the group is not formed.

A server that detects an illegal (i.e., previous or concurrent group set-up) for the same client / object pair, can forward the evidence to other members of the existing proposed group. As the request is signed by the client, a malicious server cannot falsify a bogus request. This means that it does not require a threshold of group members to convince the other group members of the illegality of the request.

8.5.2 Group updates

Group membership updates are likely to happen for one of three reasons:

1. The object owner no longer requires service from the object replication service, and closes the group down, which deletes the instance of the object from the service.
2. The object owner receives information from servers within the group of the potential corruption of other group members.
3. The object owner receiver external information that members of the group could be potentially corrupt.

[11] This can only be detected by the service if the client requests overlapping groups for the same object identifier, which is a problem also covered in the next section of this chapter.

There is very little difference between the second and third instances, except in the source of the information, and the level of integrity which the object owner is willing to attribute to the information.

Object deletion This method is the simplest, with the client putting in a legitimate request for object deletion. The honest members of the server group then remove the instance of the object from their object store, and make a note of the deletion, in case further access to the object is requested, for which a reply can be given that the object no longer exists.

Group Internal information In this instance, the object owner needs to be sure that he can justify the decision to change the group structure. The client will receive any reports about potential faulty group members from other group members. If the client regards each member with equal suspicion, then it will require a minimum of $\lfloor \frac{n-1}{3} \rfloor + 1$ accusations against a particular member for the client to act. The client can of course apply a greater weighting to the evidence of a smaller number of group members if it is willing to trust any particular group member more, but is primarily using the replication server as a means of safeguarding against a group member suffering a benign crash. The exact update policy with regard to object identifiers etc. is expanded upon below.

Group External information This scenario is similar to the one above, and uses the same update mechanisms, but the source of the information need not have the same verification principles as those mentioned above.

Controlling group update protocol

To update the object, it is important that we can update the object in place without loosing reference to the state information. This is important in case legitimate clients request information from servers which are no longer members of the controlling group for the object. Our protocol outlined below only works in the case of honest clients, as a malicious client can delete the object and re-create it is an instance of a new object without leaving any trace. Dealing with this is beyond the scope of our work[12].

[12] We conjecture that stopping such an event is impossible when clients have local manipulation of objects combined with a creation process external to the control of the server.

A group membership for an object is made up of a list of servers:

$$serv\_list = \{c_{s_1}, c_{s_2}, \ldots, c_{s_n}\}$$

Where the number of servers is n. Each object has the following information stored at each server that makes up the group:

C.I.D.	:	Client Identifier.
O.I.D.	:	Object Identifier.
A.C.I.	:	Access Control Information.
H.C.I.	:	History Control Information.

The C.I.D. identifies the client that generated the object, and the O.I.D. is a unique identifier generated by the client for each object he registers with the service. Together they form a unique identifier[13] for each object in the system. The A.C.I. is explained in section 8.6.2 of this chapter. The H.C.I. is the information concerned with updating legitimate requests, its properties and update procedure are described below.

The H.C.I. is a tuple $<list<O.I.D.>, list<serv\_list>>$ containing the information regarding the prior incarnations of the current object. The lists form a chronological order of the different object identifiers and the server lists which represented their group membership.

When a client deletes an object, he flags it as a full deletion or an update-in-place. If the request is an update-in-place, the client gives the information to the servers of the new object identifier, and new server list, which are added to the tail of the H.C.I. In the cast of a full delete, a null entry is included instead. The H.C.I. allows a server to extend the access control information across incarnation boundaries if the object owner so wishes. This can allow the server to grant access rights to an object which appears in the history of a current object.

8.6 Object Control

In this section, we outline the mechanisms used to enforce object control. First we look at maintaining secrecy in an environment where an object needs to be replicated, and secondly at the access control structure used above the replication layer.

[13] We do note however that this is unique only if the client generates a new identifier for each object. There is nothing to stop a malicious client re-using an identifier, as detecting the existence of to duplicated identifiers in a loosely coupled system without using a trusted central administrator is a difficult task.

8.6.1 Object replication

In order to maintain secrecy as part of an access control policy, we need to have control of the information needed to gain access to the object split between the various servers that make up the group. This guarantees that a single malicious server – or small subset – cannot compromise the secrecy of the object on its own.

To achieve this we make use of Shamir's *(k,n) secret sharing scheme* [Sha79] to distribute the encryption key between the group members. Our reason for doing this is efficiency as a tradeoff against perfect secrecy. We could secret-share the object itself, assuring that as long as no more than $\lfloor \frac{n-1}{3} \rfloor$ servers give up their shares[14], then the object remains secret for ever. Unfortunately, this would give as an unwieldy computational overhead for large objects. Our method relies on encrypting the object, and secret sharing the key between each of the members. This provides us with a lower computational overhead, while reducing the secrecy of the object from perfect secrecy to the best known attack on the encryption algorithm.

We now describe the message content sent to the group members. The following notation is used for the various data items included in the messages:

k_o : Object encryption key.
k_{S_i} : S_i's public key.
k_o^i : S_i's share of the object encryption key.

The content of the packet sent to the service in the client to server interaction protocol of section 8.4.2 is shown below:

$$\{Obj\}_{k_o},\ A.C.I.,\ \{h(D),\ k_o^1\}_{k_s^1},\ \{h(D),\ k_o^2\}_{k_s^2},\ \ldots,\ \{h(D),\ k_o^n\}_{k_s^n}$$

A.C.I. is the access control information, and is described in the next section. $h(D)$ is a hash of the concatenation of the encrypted object, and the access control information for integrity verification. There is little reason to encrypt the access control information, as each server needs it in order to carry out its task, which would allow a malicious server to leak the information.

The reason for sending all servers a copy of every k_o^i, rather than only sending each piece to the server that requires it, is that if a server is isolated due to severe network disruption, then it can contact the other replicas to receive the most recent object state information.

[14] Because of the nature of the *(k,n)* secret sharing scheme, this figure could be arbitrary, but is set to $\lfloor \frac{n-1}{3} \rfloor$ in order to match the requirement for the number of correct processes in a group.

We make an interesting observation regarding the use of Shamir's plain secret sharing method over a *Verifiable Secret Sharing* (VSS) technique [GM95, CGMA85, Fel87]. VSS is a more powerful abstraction, and it might be seen to be an advantage to have it over a normal secret sharing mechanism. Unfortunately we do not gain anything from using it in this scenario.

As the servers are incapable of checking the integrity of the encrypted content of the object, the servers gain nothing from knowing if a share given to them by a client is valid or not. The client who receives the shares when performing a read on the object cannot prove after the fact that it was a malicious client or malicious server that injected a false share into the system either.

8.6.2 Access Control

Access control is structured predominantly around a capability based mechanism. The object owner generates the capabilities and hands them off to other clients in the system.

The basic structure of the access control system is described in each of the four following sections, each covering the token structure, access control commands, client types, limited server storage and transfer of capabilities in turn.

Token Structure

The basic structure of the access control token is shown in figure 8.5. The token itself is signed by the owner of the object. We make use of an extension to the principle of identity based capabilities presented by Gong [Gon89].

< O.I.D., server_list, control type,

transfer flag, recipient type, [C.I.D], [G.I.D.]>

Figure 8.5: Access control token

The sections of the token and the Object Identifier, the server list, the control value of the token, a transfer / non-transfer flag and client or group identifier.

The Object Identifier, and server list are those described in previous sections. Each of the other values is described in its own section below, in addition to section describing the server storage required to manage group access to the object.

Control Commands

There are three basic control commands for objects:

Read Allows the bearer of the token to have read access over the object.

Write Allows the bearer of the token to write a new value to the object store in place of the current value.

Delete This allows the bearer to delete the object. Because of the consequence of deleting the group, and the need for updating the reference list for the object that is associated with this action it is dealt with as a different permission than a standard write permission on most file systems.

The capability can posses any mix of these three control sequences.

Bearer Types

There are two types of bearer type:

Named A named capability will contain a client identifier [15] for the client to which the capability is issued.

Group A group capability will contain the identifier of the group as stipulated by the owner of the object. How this is dealt with is described in further detail in the section below describing server storage.

Global A global capability is by its very nature transferable, and the transfer flag is set accordingly. This type of capability is generally issued for reads when the object owner wishes to allow any other client to read an object.

Server Storage

Even though we wish to use a capability based mechanism, there is a reason for storing some of the access control state at the server as well. The two types of information are:

Group control information A structure of a tuple containing a group name and a list of the client identifiers which are members of the group is stored at the server site. This is preferable to issuing the

[15] We assume a unique means of identifying a client, either globally, or from the point of view that object owners can specify clients identifiable to themselves.

group structure along with the capability, as it makes it possible for the object owner to modify the structure after the capability has been issued. When a client presents a group based capability, it has to present its client identification as well, which is checked against the current group membership.

Revocation list There are two types of information stored in the revocation list. The first is a list of client identifiers which have had their capability revoked by the owner, and the second is a notification if a global access has been revoked[16].

Transfer Flag

When the transfer flag is set in named capabilities, the client identified within it can generate another token to indicate that they are transferring the rights present in the capability to another client. This is done by issuing a new signed token containing the following information $<O.I.D., N.I.D.>$, where N.I.D. indicates the client identifier to which the rights are being passed. This can be repeated, generating an authorisation chain.

In the case of group capabilities, the client can generate the same token, but when the server comes to verifying the access, then they interrogate the identity of the original holder against the current group membership list.

In the case of global capabilities, the transfer flag is set by default.

8.7 Conclusions

In this chapter we design a global service which can provide resilient distributed access to objects, while retaining the control of the group management structure in the hands of the object owner.

We provide a new atomic broadcast algorithm that allows for process failure without having to resort to changes in group views. This is a benefit over existing group-based mechanisms that require a view change to allow the protocol to terminate. Providing for view changes in a system where secrecy is an issue would vastly complicate the liveness section of the protocol, with a need for updating the secret sharing portion of the service, potentially without any outside aid.

[16] Of the three access types, the name based and global based accesses are addressed by the revocation list, where as the group based mechanism inherently provides a revocation mechanism.

Group updates themselves are dealt with by having the client dictate the group membership and use the broadcast protocol's ability to atomically deliver messages in the face of failure to allow updates to take place.

Unfortunately, there is an inherent performance penalty in using our broadcast algorithm. Our work is related to the broadcast protocols used in *Isis* [BSS91] and *Rampart* [Rei94b], where a comparison between the message delays demonstrate that strengthening the broadcast protocols to be secure reduces their scalability greatly.

There are other mechanisms that provide for a more efficient broadcast algorithm [MMS95, MR97b], but we do not believe these to be applicable in our scenario. Their algorithms are built upon chaining mechanisms, where correct processes acknowledge messages previously seen when forming a broadcast. This allows processes to locally build directed graphs which provide a total order. Unfortunately, their improvement in throughput is measured as the delivery latency over a given number of messages, and as we do not envisage a large traffic throughput in relation to the time to answer a single query, their mechanism does not offer us a distinct advantage.

Another more lightweight replication mechanism which could be proposed for a similar service is that of quorum based systems [MR98a, MR97a, MRW97]. The difficulty of using such mechanisms for what we wish to achieve is a similar problem to the one related to group changes, in that maintaining object secrecy becomes a difficult task. This difficulty would be compounded further by the quorum-based mechanism's limited cross-sections of process groups.

Chapter 9

Conclusions and Future Work

9.1 Overview

In this chapter we overview the results obtained from the work presented in this dissertation, and draw some conclusions from our study. We also provide an overview into what we see as future work in the area given the insights gained from our work to date.

9.2 Results

In the first section of our dissertation (chapters 3 & 4), we looked at the notions of faults and failures and what they mean in terms of security protocols.

Our findings break down into two categories, which are related to the flow of execution and the flow of information respectively.

Firstly, we believe the control of the flow of execution to be an integral part of combining fault tolerance with security. In section 3.2.1, when translating the definitions of state into protocols, we end up with the message content encapsulating the state within the cryptographic protocol. This supports our notion that controlling flow is of relevance, because there subsequently needs to be an ability to monitor the existing state in order to carry out the constituent parts of fault tolerance (i.e., error detection etc.). This gets translated into secure protocols by our definitions in section 3.2 of the *beneficiary* and *tightly coupled protocols*.

Our notion that control of execution is an integral part of the ability of providing fault tolerance in secure protocols is also clearly shown in section 4.3, where we expand upon the threat model faced by principals wishing to communicate anonymously. We demonstrate that, if the vulnerability envisaged by the threat model is taken to its limit, then we are still capable of surviving an attack if we allow the principals to govern the execution of the protocol itself.

In the second instance, we believe the control of information flow to be a method of achieving the goals of fault tolerance in a security environment.

In section 3.3, we note that the information required to enable a principal to derive "beliefs" with regard to a cryptographic protocol moves from the information gained about actions of the other participants to the information gained from the voting section of the protocol. In section 3.4 we also notice that the reliance boundary moves, allowing the principal in essence to not bestow reliance directly on the members of the replicated service. Consequential to this move, the information that the client relies upon is derived from a different source.

In sections 4.2 and 4.4 we show two different methods of using this notion of controlling the flow of information to decrease the reliance on the service by increasing the amount of control available to the client.

Section 4.2 provides a new twist on a debate that has already been studied in the literature. The information that is being controlled here is something that is of use to the service (notably the freshness of the request).

We believe the example studied in section 4.4 to be of greater interest, where we demonstrate a new variant of a denial of service attack. The threat model we use is closely linked to our motivation of using the principles of fault tolerance to improve the security as seen by the client. The interesting point with regard to the control of information used in this solution, as opposed to the solution employed above, is that the information flow being controlled is that which is of use to the service and not the client. By controlling the information, the client has greater assurances with regard to the outcome of the request, while not affecting the ability of the service to carry out its duty.

It is usually a goal for the principal that uses the information necessary to carry out the secure function (e.g., crypto key) to have control over it, but what this section demonstrates, is that the control does not necessarily need to reside with the party concerned with its integrity, and that in sharing the control, two or more principals can achieve their respective goals.

In the second section of our dissertation (chapters 5 & 6), we study the effects of applying various types of fault tolerant constructions within security.

Our main goal in this section was to see how constructions other than server replication could be applied to security services. What we demonstrate by our findings, is that the ability of a fault tolerant model to be integrated within a security function without change depends on the motivation behind its application, which demonstrates that conceptual transition cannot be taken for granted.

We believe that fault tolerant models that are used to strengthen a security function that already exists within the system's design (e.g., server replication for a key server) can be integrated quite easily, without any great need for changes in the design assumptions.

Conversely, we believe that if a model is being used to implement a new security function directly (e.g., our notion of controlling the interface communication in section 5.4), it is much more likely to require some degree of change to its structure or assumptions.

In section 5.3, we look at the use of exception signalling to augment the functionality of certificate revocation systems. Our results in that section demonstrate that there is little that needs to be changed in the model, or its structure.

Conversely, in chapter 6 and sections 5.4 and 5.5, we look at using models within fault tolerance to generate new forms of functionality.

In section 5.4, we are interested in what effect recovery across an interface has on the assumptions used when designing the flow of information. When these techniques are used in their native environment, they restrict the information flow between processes that are deemed to be competing on the interface. We argue that this assumption needs to be revised in some scenarios when used for security.

In section 5.5 we demonstrate that the use of information sharing across a boundary, specifically for the act of recovery, requires some change in its application. When adapted for security, we show that the control of both the information and recovery process should be empowered to the security policy of the principal concerned with the successful recovery, which moves away from the direction of the work in its original environment.

Chapter 6 demonstrates the use of timing measurements to provide a new means for defeating timing or replay attacks.

This provides secure and insecure variants of the same resource available to nodes on a network (notably the bandwidth division). This echoes similar reserve partitions in processor design, with the distinction between *supervisor* and *user* modes, and demonstrates that we require some underlying support from the medium which we are trying to segregate in order to achieve the result.

Our mechanism is not able to stop all such attacks, but is most adept at detecting suppression type attacks. It is a side-effect of this result that we are also able to counter denial of service attacks as well as our original goal of stopping replay and timing attacks.

There might be a temptation here to say that denial of service attacks are only a subset of timing attacks [1], but we believe this not to be true. We back up our claim by the observation that not all denial of service attacks use timing, and not all timing attacks require a denial of service.

Examination of the types of protocols which our design favours (in section 6.5.2), strengthens our argument for our notion of tightly coupled protocols defined in section 3.2.1.

As an overall observation regarding this section of work, we believe the dichotomy which arises when applying fault tolerance to a security function is due to the nature of generating a new security function.

A mechanism that is used to increase the resilience of a security function that has previously been designed into the system is less likely to require modification, as it is being used to strengthen a policy that has already been considered and implemented. In effect, the security model already in place is acting as an abstraction to the fault tolerant model used beneath it.

In the examples where we use a fault tolerant principle to directly generate a new security mechanism – as in section 5.4 – we are likely to observe conflicts. This is because the fault tolerant model is used openly by the client accessing the secure mechanism and is not cleanly abstracted away, which requires the assumptions used when initially generating the model to be reevaluated.

In the final section of our dissertation (chapters 7 & 8), we look at differing mechanisms for server replication. The driving force behind existing designs is the increase in resilience of the service.

In chapter 7, we use primary backup replication, which provides a more lightweight form of replication than state machine. Primary backup can be of use in scenarios where the threat model is not a fully byzantine one, and in chapter 8 we provide the design for a server that, even in a full byzantine scenario, allows for client control over the group membership. This allows the client to dictate its own policy when it comes to trusting various parts of a replicated service.

We demonstrate in chapter 7 that it is possible to use a different replication mechanism – which uses a more lightweight technique – for scenarios

[1] Evidence that would back up this claim can be seen in the *suppress-replay* attack of Gong [Gon92], which uses an initial denial of service to facilitate a replay attack within the timing window of the Kerberos authentication mechanism. While this highlights a direct link between the two, we believe them to be related and not subsets of each other.

where the threat model does not include an attack that leaves the server in a byzantine failed state.

What existing work there is on securing servers in a threat model which does not extend to a byzantine failed server, uses the state machine approach, and like the work in *Rampart* focuses more on the design from the server's point of view rather than that of the client.

What we can demonstrate in chapter 7, is that a replication mechanism can be generated such that the strength of the trust relationships between the replicas can be dictated by the security policy of the client.

In chapter 8 we provide the design for a group replication mechanism that is built above an atomic broadcast protocol. Our new atomic broadcast algorithm does not need group updates to avoid the impossibility result of Fischer et al. [FLP85].

Our new atomic broadcast protocol uses the notion of failure detectors [CT96] coupled with earlier work on randomness in consensus protocols [BT85] in order to achieve this result. There is a performance penalty from using State Machine replication in such systems, but other options such as quorum based systems [MR97a] would make update of objects that require secrecy a complicated task.

What we aimed to achieve, and show in these two chapters is that it is possible to give the client control over the replication mechanism. It is unfortunate, but not surprising that such mechanisms increase the cost of the algorithms needed to implement them.

An interesting point for potential work would be to search for more lightweight means of giving the client control over replication schemes. We believe that in widely distributed systems such as the Internet, increasing the reliability of security services should be a common goal. Notably that giving the client some degree of control over the actions of processes which it is less likely to trust than in a local network should be studied further.

9.3 Conclusions

We now look at some conclusions that can be derived from our work which are not dealt with explicitly in the findings of the previous section. These results are not directly obtained from the initial motivation for our research, but provide some insight into the problems faced by further work in this field.

Our first observation is based loosely on where in the OSI model the functionality is placed, and is an extension of our notion of control.

Our work in chapter 6 provides timing measurements for increasing protocol resilience to timing and replay attacks. It would be impossible for the client to use this mechanism without trusting the machine which connects them to the network. It is already well known that carrying out a secure function on a machine that the client does not trust is futile. What we promote here in addition, is that in our design, due to the nature of controlling the physical connection, it would be impossible for the client itself to control the fault tolerant functionality without the aid of the machine on which they are situated.

With the terminology of the OSI model in mind, if the security design includes some fault tolerant functionality that requires the aid of the layers below the application layer, then it requires the client to share the control of the security protocol with the layers below. By sharing this control, it moves away from our ideal scenario in which the client controls the whole protocol.

We differentiate this from the need to trust the host on which the client is situated. As mentioned above, the host needs to be trusted in any case, but part of our goal in this dissertation was to provide control of the fault tolerant mechanism to the client. In our observation above we note that this might not be possible for reasons that have nothing to do with the trust relationship with the other parties directly involved in the protocol. Thus it might be entirely feasible for the trust relationship in the protocol to allow for the client to receive more control, but that the physical implementation prevents it, which is something we did not initially consider.

This observation might initially seem superfluous, as in our design, the client has to trust the node on which it resides, but we claim it to be of importance by highlighting possible future implementations, where the client is logged on to a workstation, and the security function of the protocol is carried out on a smart card connected to the workstation. In this scenario the client does not necessarily trust the workstation, in which case, it would be difficult to justify allowing the workstation to share control in the fault tolerant part of a security protocol.

A second observation related to the first one, is that the fault tolerant component of the design can be shared (e.g., in chapter 6 it is shared with the host) while still not compromising the security functionality of the system. This demonstrates that the security and fault tolerance portions of the system design can be managed separately, even when integrated to achieve improvements in the security service, which we believe to be an interesting point.

A third observation we make is that of a parallel between our ability to provide more lightweight replication using primary backup (in chapter 7) and more recent work carried out by Reiter and Malkhi on quorum based systems for secure replication [MR98a, MR98b].

In their work on quorum based systems, they point out that secure replication using the state machine approach is not generally scalable. Their work is scalable due to the lack of need for server to server communication. The server to server communication is something that is also minimised in our work on primary backup. Their work does however require server to server communication to allow a state update, which increases the protocol complexity. Their work on consensus objects [MR98b] does not require server to server communication, but does require several rounds of communication where the client forwards messages between servers, which has the same overhead as a server to server protocol with one server acting as a relay.

The relation between our work and quorum based systems is highlighted by the fact that reducing the functionality within the replication – in our case a reduction of the byzantine threat model, and in theirs reducing the main operation to a read – allows a reduction in the complexity inherent in a full state machine approach. It would be interesting to see to which extent different types of reduction in functionality can reduce the overhead of the replication mechanism.

9.4 Future Work

In this section, we briefly outline areas of cross-over between fault tolerance and security that we believe could provide other avenues of research given the experience in our current work.

The first area we cover is that of timing in distributed computations. In work done by others on primary backup approaches to server replication in benign failure environments, one of the things studied [DGMS85, Ske82] is allowing servers not within the primary partition group to provide service, and then using an update merge policy to reconcile differences when the partitioning ends. It would be interesting to see how such work could be carried across to a secure environment, where byzantine corruption of the servers is not the primary threat model, but where the service does provide some security relevant functionality. What type of secure services could be built to function on possibly divergent information, and still provide some degree of service?

An inherent problem in asynchronous distributed computation in a byzantine environment is the use of consensus to achieve virtual synchrony. The work of Dolev et al. [DDS87] – which is closely related to the work of Dwork et al. [DLS88] on partial synchrony – provides a study of minimal levels of synchrony needed with regard to various parameters of the system (namely processor, communication, message order and transmission mechanism), and how each affects the ability to achieve consensus. It would be interesting to study protocols that achieve consensus under the different parameters in a byzantine environment, and measure their trade-offs with regard to real world estimates of methods of supplying the parameters.

In their work on using quorum systems, Malkhi and Reiter [MR98a, MR98b] the rules regarding virtual synchrony are relaxed in order to provide a more lightweight replication mechanism, while still allowing for a fully byzantine environment. It would be interesting to find security functions which could still provide service without a global sense of synchrony, but replaced by the use of a client defined view of synchrony. A simple example of this is the work done by Blaze [Bla96].

A second area we believe could provide interesting research is that of failure detectors. In their work on unreliable failure detectors Malhki and Reiter [MR97c] note that providing a fuller extension of the work carried out by Chandra and Toueg [CT96] for a byzantine environment would be useful.

Our work in chapter 8 only uses the *Completeness* property of failure detectors [CT96], and does not touch on the use of *Accuracy*, because we note the inherent difficulty of maintaining the state needed to achieve accuracy in a loosely coupled distributed environment. There are two forms of order in multicast protocols, local and global order [HT94]. It would be interesting to study what effect trying to implement the two different primitives in a byzantine environment would have on the properties needed for the failure detectors.

A third area of research, not directly covered in our current work, is the use of hardware mechanisms to improve the resilience and efficiency of distributed server computations.

One of the problems with providing for distributed computation on the Internet, is the inherent risk that a site could be hacked and the server corrupted. If we were to use hardware at each of the sites to provide the core security functionality, then all an externally compromised machine could do would be to emulate a benignly failed machine.

Siu et al. [SCY98] provide a design for a mixed failure environment. If we accept that some portion of corrupt servers could be directly under the control of malicious entities, and another portion can only be compromised from the outside, can we use the principles inherent in their mixed failure mode design to increase the resilience of a distributed server on the Internet?

A further area of interest would be to study how best to design more generic security protocols (e.g., authentication, key exchange etc.) and how to make them recoverable from compromise. There is a small amount of work done in this area already [HK95], and generalising it could prove to be fruitful.

Bibliography

[AD76] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the* 12^{th} *International Conference on Software Engineering*, pages 562–570, October 1976.

[AL81] T. Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall International, 1981.

[AL82] T. Anderson and P.A. Lee. Fault tolerance terminology proposals. Technical Report 174, University of Newcastle upon Tyne, Computing Laboratory, University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England., April 1982.

[ALS78] Thomas Anderson, Peter A. Lee, and Santosh K. Shrivastava. A model of recoverability in multilevel systems. *I.E.E.E. Transactions on Software Engineering*, SE-4(6):486–494, November 1978.

[ALS79] T. Anderson, P.A. Lee, and S.K. Shrivastava. *System Fault Tolerance*, chapter 5, pages 153–210. Cambridge University Press, 1979.

[And94] Ross J. Anderson. Why cryptosystems fail. *Communications of the A.C.M.*, 37(11):32–40, November 1994.

[And96] Ross J. Anderson. The eternity service. In *Proceedings of Pragocrypt '96*, October 1996.

[Aur97] Tuomas Aura. Strategies against replay attacks. In *Proceedings of the* 10^{th} *Computer Security Foundations Workshop*, pages 59–68, June 1997.

[BAN89] M. Burrows, M. Abadi, and R.M. Needham. A logic of authentication. *Proceedings of the Royal Society of London*, 246:233–271, 1989.

[Ber96] John Beric. Real security. Mondex Magazine, Summer 1996. pp. 5–7.

[Bla96] Matt Blaze. Oblivious key escrow. In Ross Anderson, editor, *Proceedings of the First International Workshop on Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*, pages 335–343. Springer-Verlag, May/June 1996.

[BM92] Navin Budhiraja and Keith Marzullo. Tradeoffs in implementing primary-backup protocols. Technical Report TR 92-1307, Department of Computer Science, Cornell University, 1992.

[BMST92] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary-backup protocols. In *Proceedings of Distributed Algorithms* 6^{th} *International Workshop*, volume 647 of *Lecture Notes in Computer Science*, pages 363–378, November 1992.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. *Distributed Systems*, chapter 8, pages 199–216. Addison-Wesley, 1993. 2^{nd} Edition.

[BO83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second Annual A.C.M. Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[BO85] Michael Ben-Or. Fast asynchronous byzantine agreement. In *Proceedings of the fourth Annual A.C.M. Symposium on Principles of Distributed Computing*, pages 149–151, 1985.

[BSS91] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *A.C.M. Transactions on Computer Systems*, 9(3):272–314, August 1991.

[BT83] Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the Second Annual A.C.M. Symposium on Principles of Distributed Computing*, pages 12–26, August 1983.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[CAR83] R. H. Campbell, T. Anderson, and B. Randell. Practical fault tolerant software for asynchronous systems. Technical Report 187, University of Newcastle upon Tyne, Computing Laboratory, University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England., August 1983.

[CD89] Benny Chor and Cynthia Dwork. Randomization in byzantine agreement. *Advances in Computing Research*, 5:443–497, 1989.

[CGMA85] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Proceedings of the* 26^{th} *I.E.E.E. Symposium on Foundations of Computer Science*, pages 383–395, October 1985.

[CH96] Bruce Christianson and William S. Harbison. Why isn't trust transitive? In Mark Lomas, editor, *Proceedings of Security Protcols International Workshop*, volume 1189 of *Lecture Notes in Computer Science*, pages 171–176. Springer-Verlag, April 1996.

[Cha81] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the A.C.M.*, 24(2):84–88, February 1981.

[Cha82] David Chaum. Blind signatures for untraceable payments. In *Proceedings of Crypto '82*, pages 199–203, 1982.

[Cha88] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[CHTCB95] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. Impossibility of group membership in asynchronous systems. Technical Report TR95-1533, Cornell University, Department of Computer Science, August 1995.

[CL96] Bruno Crispo and Mark Lomas. A certification scheme for electronic commerce. In *Proceedings of the Security Protocols*

International Workshop, volume 1189 of *Lecture Notes in Computer Science*, pages 19–32, April 1996.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CU95] H. Jonothan Chao and Necdet Uzun. An ATM queue manager with multiple delay and loss priorities. *A.C.M. Transactions on Networking*, 3(6):652–659, December 1995.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the A.C.M.*, 34(1):77–97, January 1987.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *A.C.M. Computing Surveys*, 17(3):241–370, September 1985.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the A.C.M.*, 35(2):288–323, April 1988.

[DR86] J.E. Dobson and B. Randell. Building reliable secure computing systems out of unreliable insecure components. Technical Report 214, University of Newcastle upon Tyne, Computing Laboratory, University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England., March 1986.

[DS81] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the A.C.M.*, 24(8):533–536, August 1981.

[Fel87] Paul Feldman. A practiacal scheme for non-interactive verifiable secret sharing. In *Proceedings of the* 28th *Annual Symposium on the Foundations of Computer Science*, pages 427–437, 1987.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FM85] Paul Feldman and Silvio Micali. Byzantine agreement in constant expected time (and trusting no one). In *I.E.E.E. Symposium on Foundations of Computer Science*, pages 267–276, 1985.

[GB94] Annie Gravey and Soren Blaabjerg, editors. *Cell Delay Variation in ATM Networks.* COST 242 Mid-term Seminar Interim Report, August 1994. Sections 1 & 2.

[GG94] Li Gong and Jack Goldberg. Implementing fault-tolerant sevrvices for hybrid faults. Technical Report [SRI-CSL-94-04R], S.R.I. International Computer Science Laboratory, 1994.

[GGL93] Jack Goldberg, Ira Greenberg, and Thomas F. Lawrence. Adaptive fault tolerance. In *Proceedings of the I.E.E.E. workshop on Advances in Parallel and Distributed Systems*, pages 127–132, October 1993.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the* 7^{th} *symposium on Operating Systems Principles*, pages 150–159, 1979.

[Gli86] Virgil D. Gligor. On denial-of-service in computer networks. In *Proceedings of the International Conference on Data Engineering*, pages 608–617, February 1986.

[GM95] Rosario Gennaro and Silvio Micali. Verifiable secret sharing as secure computation. In *Advances in Cryptology - Eurocrypt '95*, volume 921 of *Lecture Notes in Computer Science*, pages 168–182, May 1995.

[Gol98] Dieter Gollmann. Insider fraud. In Bruce Christianson, Bruno Crispo, William S. Harbison, and Michael Roe, editors, *Proceedings of the* 6^{th} *International Workshop on Security Protocols*, volume 1550 of *Lecture Notes in Computer Science*, pages 213–219. Springer-Verlag, April 1998.

[Gon89] Li Gong. A secure identity-based capability system. In *Proceedings of the 1989 I.E.E.E. Symposium on Security and Privacy*, pages 56–63. I.E.E.E. Computer Society, technical committee on Security and Privacy, in cooperation with the International Association for Cryptologic Research (IACR), I.E.E.E. Computer Society Press, May 1989.

[Gon92] Li Gong. A security risk of depending on synchronized clocks. *A.C.M. Operating Systems Review*, 26(1):49–53, January 1992.

[Gon93a] Li Gong. Increasing availability and security of an authentication service. *I.E.E.E. Journal on selected areas in communications*, 11(5):657–662, June 1993.

[Gon93b] Li Gong. Variations on the themes of message freshness and replay. In *Proceedings of the Computer Security Foundations Workshop VI*, pages 131–136, June 1993.

[Gon94] Li Gong. Fail-stop protocols: An approach to designing secure protocols. Technical Report SRI-CSL-94-14, S.R.I. International Computer Science Laboratory, S.R.I. International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, October 1994.

[Gra78] J. N. Gray. *Notes on Data Base Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, chapter 3.F, pages 393–481. Springer-Verlag, 1978.

[GS95] Li Gong and Paul Syverson. Fail-stop protocols: An approach to designing secure protocols. In *Preprints of the* 5^{th} *International Working Conference on Dependable Computing for Critical Applications*, pages 44–55, September 1995.

[GS97] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, April 1997.

[Hal] Tim Hale. Integrating ATM backbones with high-speed Ethernet at the edge. http://www.3com.com/technology/tech_net/white_papers/500668.html. 3com White Paper.

[HK95] Tzonelih Hwang and Wei-Chi Ku. Reparable key distribution protocols for internet environments. *I.E.E.E. Transactions on Communications*, 43(5):1947–1949, May 1995.

[HS91] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[JvRS96] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Supporting broad internet access to TACOMA. In *Proceedings of the Seventh A.C.M. Sigops European Workshop*, pages 55–58, September 1996.

[Koc] Paul Kocher. A quick introduction to certificate revocation trees. http://www.valicert.com/company/crt.html.

[Lam89] Leslie Lamport. The part-time parliament. Technical Report 49, Digital Equipment Corporation Systems Research Centre, September 1989.

[LB92] Kwok-Yan Lam and Thomas Beth. Timely authentication in distributed systems. In *Proceedings of European Symposium on Research in Computer Security*, volume 648 of *L.N.C.S.*, pages 293–303, November 1992.

[LBF$^+$93] Bev Littlewood, Sarah Brocklehurst, Norman Fenton, Peter Mellor, Stella Page, David Wright, John Dobson, John McDermid, and Dieter Gollmann. Towards operational measures of computer security. *Journal of Computer Security*, 2:211–229, 1993.

[LG90] Shyh-Wei Luan and Virgil D. Gligor. On replay detection in distributed systems. In *Proceedings of the* 10th *International Conference on Distributed Computing*, pages 188–195, May 1990.

[LGSN89] T. Mark A. Lomas, Li Gong, Jerome H. Saltzer, and Roger M. Needham. Reducing risks from poorly chosen keys. In *Proceedings of the* 12th *A.C.M. Symposium on Operating Systems Principles*, December 1989. Published as *A.C.M. Operating Systems Review*, 23(5):14–18.

[Lit79] Bev Littlewood. How to measure software reliability and how not to. *I.E.E.E. Transactions on Reliability*, R-28(2):103–110, June 1979.

[LM95] Ian M. Leslie and Derek R. McAuley. ATM: Theory and practice. University of Cambridge Computer Laboratory lecture slides in bound volume, 1995.

[Lom] Mark Lomas. Personal Communication.

[LSP82] Leslie Lamport, Robert Shostak, and Marchall Pease. The byzantine generals problem. *A.C.M. Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Mea94] Catherine Meadows. The need for a failure model for security. In *Proceedings of the* 4^{th} *International Workshop Conference on Dependable Computing for Critical Applications*, 1994.

[Mea95] Catherine Meadows. Applying the dependability paradigm to computer security. In *Proceedings of the 1995 New Security Paradigms Workshop*, 1995.

[MHS89] Timothy Mann, Andy Hisgen, and Garret Swart. An algorithm for data replication. Technical Report 46, Digital Equipment Corporation Systems Research Centre, June 1989.

[Mic96] Silvio Micali. Efficient certificate revocation. Technical Memo MIT/LCS/TM-542b, Laboratory for Computer Science, Massachusetts Institute of Technology, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MASS 02139, March 1996.

[Mit89] C. Mitchell. Limitations of challenge-response entity authentication. *Electronics Letters*, 25(17):1195–1196, August 1989.

[MMS95] Louise E. Moser and P. M. Melliar-Smith. Total order algorithms for asynchronous byzantine systems. In *Proceedings of the* 9^{th} *International Workshop on Distributed Algorithms (WDAG '95)*, volume 972 of *Lecture Notes in Computer Science*, pages 242–256, September 1995.

[MR77] Philip M. Merlin and Brian Randell. State restoration in distributed systems. In *Proceedings of the* 8^{th} *international symposium on Fault-Tolerant Computing*, pages 129–134, 1977.

[MR97a] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the* 29^{th} *A.C.M. Symposium on Theory of Computing*, pages 569–57, May 1997.

[MR97b] Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113–127, 1997.

[MR97c] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computuations. In *Proceedings of the* 10^{th} *I.E.E.E. Computer Security Foundations Workshop*, pages 116–124, June 1997.

[MR98a] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in phalanx. In *Proceedings of the* 17^{th} *I.E.E.E. Symposium on Reliable Distributed Systems*, pages 51–58, October 1998.

[MR98b] Dahlia Malkhi and Michael K. Reiter. Survivable consensus objects. In *Proceedings of the* 17^{th} *I.E.E.E. Symposium on Reliable Distributed Systems*, pages 271–279, October 1998.

[MRW97] Dahlia Malkhi, Michael Reiter, and Avishai Wool. The load and availability of byzantine quorum systems. In *Proceedings of the* 16^{th} *A.C.M. Symposium on Principles of Distributed Computing*, pages 249–257, August 1997.

[MvRS96] Yaron Minsky, Robbert van Renesse, and Fred B. Schneider. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the Seventh A.C.M. Sigops European Workshop*, pages 109–114, September 1996.

[Nee94] Roger M. Needham. Denial of service: An example. *Communications of the A.C.M.*, 37(11):42–46, November 1994.

[NN98] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *Proceedings of the seventh USENIX Security Symposium*, pages 217–228, January 1998.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the A.C.M.*, 21(12):993–999, December 1978.

[NS87] R.M. Needham and M.D. Schroeder. Authentication revisited. *A.C.M. Operating Systems Review*, 21(1):7, January 1987.

[NT94] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *I.E.E.E. Communications Magazine*, 32(9):33–38, September 1994.

[OP96] S. O'Connell and A. Patel. A model for the detection of the message stream delay attack. In Sokratis K. Katsikas and Dimitris Gritzalis, editors, *Proceedings of the I.F.I.P* 12^{th} *International Conference on Information Security*, pages 438–451, May 1996.

[OR87] Dave Otway and Owen Rees. Efficient and timely mutual authentication. *A.C.M. Operating Systems Review*, 21(1):8–10, January 1987.

[Per84] Kenneth J. Perry. Randomized byzantine agreement. Technical Report TR 84-595, Cornell University, Department of Computer Science, March 1984.

[PW87] Andreas Pfitzmann and Michael Waidner. Networks without user observability. *Computers and Security*, 6(2):158–166, April 1987.

[Rab83] Michael O. Rabin. Randomized byzantine generals. In *Proceedings of the* 24^{th} *I.E.E.E. Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[RBG92] Michael Reiter, Kenneth Birman, and Li Gong. Integrating security in a group oriented distributed system. In *Proceedings of the 1992 I.E.E.E. Symposium on Research in Security and Privacy*, pages 18–32. I.E.E.E. Computer Society, technical committee on Security and Privacy, in cooperation with the International Association for Cryptologic Research (IACR), I.E.E.E. Computer Society Press, May 1992.

[Rei94a] Michael K. Reiter. The rampart toolkit for building high-integrity services. In K. P. Birman, F. Mattern, and A. Schiper, editors, *International Workshop on Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110, September 1994.

[Rei94b] Michael K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the* 2^{nd} *A.C.M. Conference on Computer and Communications Security*, pages 68–80, November 1994.

[Rei96a] Michael K. Reiter. Distributing trust with the rampart toolkit. *Communications of the A.C.M.*, 39(4):71–74, April 1996.

[Rei96b] Michael K. Reiter. A secure group membership protocol. *I.E.E.E. Transactions on Software Engineering*, 22(1):31–42, January 1996.

[RG95] Michael Reiter and Li Gong. Securing causal relationships in distributed systems. *The Computer Journal*, 38(8):633–642, 1995.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the A.C.M.*, 21(2):120–126, February 1978.

[SC98] Felix Stalder and Andrew Clement. Exploring policy issues of electronic cash: The mondex case. Working Paper No. 8, July 1998. Faculty of Information Studies, University of Toronto.

[Sch84] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *A.C.M. Transactions on Computer Systems*, 2(2):145–154, May 1984.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *A.C.M. Computing Surveys*, 22(4):299–319, December 1990.

[Sch97] Fred B. Schneider. Towards fault-tolerant and secure agentry. Technical report, Cornell University, Department of Computer Science, 1997.

[SCY98] Hin-Sing Siu, Yeh-Hao Chin, and Wei-Pang Yang. Byzantine agreement in the presence of mixed faults on processors and links. *I.E.E.E. Transactions on Parallel and Distributed Systems*, 9(4):335–345, April 1998.

[SGR97] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. Anonymous connections and onion routing. In *Proceedings of the 1997 I.E.E.E. Symposium on Security and Privacy*, pages 44–54, May 1997.

[Sha79] Adi Shamir. How to share a secret. *Communications of the A.C.M.*, 22(11):612–613, November 1979.

[SK96] Bruce Schneier and John Kelsey. Automatic event-stream notarization using digital signatures. In Mark Lomas, editor, *Proceedings of the Security Protocols International Workshop*, volume 1189 of *Lecture Notes in Computer Science*, pages 155–169, April 1996.

[Ske82] Dale Skeen. On network partitioning. In *Proceedings of the I.E.E.E. Computer Software and Applications Conference*, pages 454–455, November 1982.

[SKK$^+$97] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on tcp. In *Proceedings of the 1997 I.E.E.E. Symposium on Security and Privacy*, pages 208–223. I.E.E.E. Computer Society, technical committee on Security and Privacy, in cooperation with the International Association for Cryptologic Research (IACR), I.E.E.E. Computer Society Press, May 1997.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *A.C.M. Transactions on Computer Systems*, 1(3):222–238, August 1983.

[SV97] Arun K. Somani and Nitin H. Vaidya. Understanding fault tolerance and reliability. *Computer*, 30(4):45–50, April 1997.

[Syv94] Paul Syverson. A taxonomy of replay attacks. In *Proceedings of Computer Security Foundations Workshop VII*, pages 187–191, 1994.

[Syv97] Paul F. Syverson. A different look at secure distributed computation. In *Proceedings of the* 10^{th} *I.E.E.E. Computer Security Foundations Workshop*, pages 109–115, June 1997.

[TH86] R. Turn and J. Habibi. On the interactions of security and fault-tolerance. In *Proceedings of the* 9^{th} *NBS/NCSC National Computer Security Conference*, pages 138–142, 1986.

[U.S95] U.S. National Institute of Standards and Technology. *A Public Key Infrastructure for U.S. Government Unclassified but Sensitive Applications*, September 1995.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the A.C.M.*, 39(4):76–83, April 1996.

[Wai89] Michael Waidner. Unconditional sender and recipient untraceability in spite of active attacks. In J-J. Quisquater and J. Vandewalle, editors, *Proceedings of EUROCRYPT '89*, volume 434 of *Lecture Notes in Computer Science*, pages 302–319, April 1989.

[Yah98] Raphael Yahalom. Optimistic trust with realistic *e*Nvestigators. In Bruce Christianson, Bruno Crispo, William S. Harbison, and Michael Roe, editors, *Proceedings of the* 6^{th} *International Workshop on Security Protocols*, volume 1550 of *Lecture Notes in Computer Science*, pages 193–202. Springer-Verlag, April 1998.