Using CSP to Verify Security-Critical Applications

Gordon Thomas Rohrmair
St Catherine's College

Thesis submitted for the degree of Doctor of Philosophy

Hilary Term 2005

Using CSP to Verify Security-Critical Applications

Gordon Thomas Rohrmair, St. Catherine's College
Hilary Term, 2005

Abstract

This thesis demonstrates how one can model and analyse security relevant processes using the process algebra Communicating Sequential Processes (CSP) [Hoa85, Ros98b] and its model checker FDR[1] [GGH+00]. We focus on two increasingly important areas — Intrusion detection and trusted computing.

An Intrusion Detection System is a system that detects abuses, misuses and unauthorised uses in a network. We show that our analysis can be used to discover attack strategies that can be used to blind Intrusion Detection Systems, even a hypothetically perfect one that knows all weaknesses of its protected host. We give an exhaustive analysis of all such attack possibilities that are based on our models and we discuss prevention techniques.

The second part of the thesis focuses on the verification process of trusted platforms and their environment. There exist various approaches how one can build a trusted platform. We will focus on the approaches that were suggested by the Trusted Computing Platform Alliance [TCPA03d, TCPA02] and Microsoft [Mic02, CJPL02]. We show how one can use a CSP based analysis to verify certain parts of the trusted computing architecture. In particular we will focus on authorisation protocols, session caching mechanisms and the generation of chains of trust within and outside the trusted architecture. Finally, we will use our technique to analyse digital rights management protocols that use trusted computing techniques.

To perform our verifications we have to prune away certain details of our systems. This leaves room for introducing a false sense of security. If we fail to find an attack, it is sometimes unclear whether that is an artifact of the abstractions used in the model, or whether there really is no attack. Data-independence within the CSP framework was invented by Roscoe and Lazić [Laz97]. It allows us to project systems with an infinite state space onto a finite one. We use results from this field to verify whether our analysis is complete.

[1] FDR is a commercial product of Formal Systems (Europe) Ltd.

Acknowledgements

I would like to thank my supervisor Dr. Gavin Lowe for many inspiring discussions, for collaborating with me on various papers [RL02, RL03, RL04] and for introducing me to the field of formal verification. I am also grateful for his support in my application for the Leatherseller's and Dstl scholarship.

I am grateful to everybody at the Concurrency group for giving me the opportunity to participate in great discussions. I am especially indebted to Dr. Philippa Hopcroft, Rob Delicata and Eldar Kleiner for not only proof-reading my thesis, but also for the many fruitful discussions we have had.

Many thanks are due to Dr. Jeff Sanders and Dr. Michael Goldsmith for being the examiners for my confirmation of status thesis. Additionally, Dr. Jeff Sanders has always had an open ear for my problems whether they were of academic or of another nature. He gave me many valuable insights.

I would like to thank Dr. Tom McCutcheon from Dstl for having faith in my research and supporting me financially.

I thank Holger Witte for supporting me in various technical ways, Jonathan Yonan for being my communication officer to the Great Lord himself and, of course, Ashley Arensdorf for improving my written english and for making Cowley Road 253 such a hilarious place. At this point I have to apologise to Ashley for not being able to bring light to the extremely difficult question of whether HK's P2000 or Sig's P228 is the better choice for him — I hope he can forgive me.

And last, but not least, I want to thank my parents, my sister (that little devil) and Frl. Briel for their support and encouragement.

This thesis was funded by a doctorate scholarship from Dstl (St Andrew's Road, Malvern, Worcester, WR14 3PS) and the Leatherseller's scholarship of St Catherine's. I wish to offer a warm dept of gratitude for their support.

Declaration

Hereby, I declare that I have done this work by my own and that I did not use any other help than that I have stated.

Gordon Thomas Rohrmair
Dasing, 1. November 2005

Table 1: Acronyms

AACP	Asynchronous Authorisation Change Protocol
ADIP	Authorisation Data Injection Protocol
ADCP	Authorisation Data Change Protocol
AES	Advanced Encryption Standard
BBB	BIOS-Boot-Block
BIOS	Basic Input Output System
CA	Certification Authority
CBC	Cipher Block Chaining
CD	Compact Disk
CE	Conformance Entity
CIDF	Common Intrusion Detection Framework
CPU	Central Processing Unit
CRTM	Core Root Trusted Measurement
CSP	Communicating Sequential Processes
CSS	Content Scramble System
DES	Data Encryption Standard
DIR	Data Integrity Register
DLL	Dynamic Link Library
DMZ	Demilitarized Zone
DRM	Data Rights Management
FDR	Failures Divergences Refinement
FIFO	First In First Out
FO	Fragment Offset
GUI	Graphic User Interface
HIDS	Host Intrusion Detection Protocol
HMAC	Hashed Message Authentication Code
I/O	Input / Output
ID	Identities
IDS	Intrusion Detection Protocol
IIK	Initial Intruder Knowledge
IKE	Internet Key Exchange
IPSEC	IP Security Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
KISS	Keep It Small and Simple
LAN	Local Area Network
LHS	Left Hand Side
MAN	Metropolitan Area Network
MF	More Fragments
NCA	Nexus Computing Agents
NIDS	Network Intrusion Detection Protocol

Table 2: Acronyms

NGSCB	Next Generation Secure Computing Base
NoEqT	No Equality Testing
OI-AP	Object Independent Authorisation Protocol
OS	Operating System
OS-AP	Object Specific Authorisation Protocol
P-CA	Privacy Certification Authority
PCR	Program Control Register
PE	Platform Entity
PKI	Private Key Infrastructure
PP	Protection Profile
PVP	Parameterized Verification Problem
RAM	Random Access Memory
RHS	Right Hand Side
RFC	Request For Comment
RMS	Rights Management System
RNG	Random Number Generator
ROM	Read Only Memory
RTM	Root of Trust for Measurement
RTS	Root of Trust for Storing
S-MIME	Secure Multipurpose Internet Mail Extensions
SHA-1	Secure Hash Algorithm 1
SPAN	Configuring the Catalyst Switched Port Analyzer
SRK	Storage Root Key
SSC	Security Support Component
SSL	Secure Socket Layer
TA	Trusted Agent
TCB	Trusted Computing Block
TCG	Trusted Computing Computer
TCSP	Timed CSP
TCP	Transport Control Protocol
TCP	Trusted Computing Platform
TCPA	Trusted Computing Platform Association
TDES	Triple Data Encryption Standard
TLS	Transport Layer Security
TOR	Trusted Operating Root
TPM	Trusted Platform Module
TPME	Trusts Platform Module Entity
TPS	Trusted Platform System
TSS	Trusted Sub System
TTL	Time To Live
VE	Validation Entity
VPN	Virtual Private Network
WAN	Wide Area Network

For

BM — who has been a beacon in the darkest hour even when her own light faded.

Contents

Chapter 1

Introduction

The research presented in this thesis centres around the field of computer security. In today's Internet driven world, computer security has become a facet of daily life, a prerequisite for protection in daily communication and transactions. Research in this field is critical, as poor security continues to be a major deterrent in e-commerce, both in on-line banking and shopping. A famous example is the break-in at the World Economic Forum [Sym01], where the credit card numbers of Bill Clinton and Yasser Arafat were stolen.

While it is essential for all sites to try to keep out trespassers, the survey [Utt96] observes that only 4 percent of attacks against US government computers were detected. Furthermore, only in about 1 percent of these breaches did a security officer respond to the incident. A few years ago the cracking of a site required a lot of time and know-how. Nowadays, where many crackers boast their exploitations by making the attack programmes available on the Internet, intrusion detection has become increasingly important [JM00].

Another approach to improve the security is to introduce reliable chains of trust. The core of these chains is a hardware component that can give remote entities trustworthy information about the security of the platform with which they are about to interact.

1.1 Intrusion detection systems

An Intrusion Detection System is used to detect abuses, misuses and unauthorised uses in a network, caused by either insiders or outsiders. These systems identify intrusions by spotting known patterns or by revealing anomalous behaviour of protected resources (e.g., network traffic or main memory usage).

Verification problems within the Intrusion Detection area. Ideally, intrusion detection systems have to keep track of every event in the network. This clearly results in a complex architecture. The complexity is even further increased

by the fact that today's networks are becoming more and more distributed; hence more agents have to be deployed to monitor the protected system. These agents have to communicate with each other. Furthermore, they have to be highly interactive, not only with their closest environment, but also with more centralised stations to update their knowledge base. This is of special significance, since the knowledge base represents the foundation for every agent's decision-making process.

Additionally, the malicious activities that should be detected are becoming more and more complex; for example the distributed attacks presented in [Dit05]. These attacks require agents to continuously monitor information of other agents. The future promises even more complicated systems, since they are expected to learn to detect novel attacks by employing complex self-learning algorithms.

Today's testing methods can be summarised under the category *trial–and–error*, defined as a set of predefined attack routines that are launched against a test site. Upon completion, the IDS log files reveal whether all attacks were detected. There exists a broad range of tools such as [Nes, FS90, SN98] or more specific tools [RFP02], that examine whether or not it is possible to encode a known attack in such a way that the IDS can not detect it anymore.

The advantages of the *trial–and–error* method are:

1. they retrieve quick results;
2. they are (usually) quick and easy to set up;
3. the results obtained are (usually) easy to use.

On the other hand this approach falls short on the following important points:

1. they are largely unable to find more complex attacks such as emergent faults [AKS96][1];
2. they are usually unable to find new attacks;
3. they can only be used to verify already existing programs. Thus, concepts or ideas lacking a concrete prototype cannot be verified with such techniques. This renders them utterly useless for the purpose of verifying architectures in the earlier stages of development.

A more desirable approach is one that is able to eradicate these vital disadvantages as well as satisfying the advantages listed above. More information on testing IDSs in the real world setting can be obtained from [Ran01].

[1] An emergent fault is defined as an specification violation that occurs because of unindented interaction between processes, whereas the participating processes individually satisfy their own security specification.

1.2 Trusted Computing

A trusted platform is a computing device that can communicate electronically with other devices, and that includes a non-compromisable element that functions as a foundation of trust for its platform. It allows the user to monitor the state of the system and to enforce a security policy upon the collected information. In addition, it guarantees other users a certain degree of trust.

Verification problems within the trusted computing area Solutions that can forge chains of trust within a platform have to keep track of every security relevant event that takes place. Looking at today's computer architectures it becomes obvious that many interactions between various parts of these systems occur. The fact that operating systems have to provide more and more functionality and that hardware has to offer more and more services increases this trend for the foreseeable future. Subsequently the logging and reporting mechanisms that should present reliable information about the events that take place increase as well.

Furthermore, these chains of trust should be extendable for reaching other platforms to form virtual clusters of reliable counterparts. Therefore, the security mechanisms have to interact with each other in such a way that no dishonest user can falsify the system state of his platform. Additionally, the transactions between these platforms become progressively complex. Hence, the methods that guarantee that no dishonest user can spread illicit information about his platform's system state increase in complexity.

Finally, only a few prototypes for the most prominent examples of trusted computing architectures exist at the moment. Thus, if one wants to verify these methods one is confined to the verification techniques that do not require a prototype.

1.3 Scope of this thesis

There are various approaches to analysing security relevant components in a network. However, nearly all of them have severe problems, such as: the inability to detect completely new attacks, the environment of a certain component can hide attack possibilities, these techniques can be used at the earliest after a fully functional prototype has been created. We want to develop a framework using CSP to spot complex vulnerabilities that are caused by unexpected feature and component interactions within a network. The resulting errors are often called *emergent faults*, meaning that although every component within a network is working according to a certain specification, when put together possibilities of erroneous interactions arise.

The second part of the thesis is concerned with the application of CSP-based verification techniques of security components to Trusted Computing Platforms (TCPs).

We will split these objectives into the following parts:

Firstly, we investigate whether CSP is a suitable calculus to model intrusion detection infrastructures with all its operating activities (e.g. spotting attacks). We achieve this by building small networks to spot already known attacks and then use the gathered data to extend our models to detect unknown attacks. Unfortunately, this is not possible for our TCPA analysis, since there are no well known attacks (as far as the author is aware). Therefore, we will follow a different path. We will analyse smaller parts of the trusted platform to determine whether our technique can also be applied to larger models. Once that is established, we will include more features and interactions.

Secondly, model checkers such as FDR have shown their usefulness and efficiency for modelling and verifying security relevant processes [RSG+01]. However, since FDR explores the whole state space of its models, the state space has to be finite; even more, the scopes of data types involved have to remain small, otherwise the processes reach an unmanageable complexity. This leaves us with a serious problem: it is often required to provide a supply of many different data values of some type to spot an attack, or more generally, to detect a specification violation. Consequently, after pruning away details, we can never be certain whether the real-world system is absolutely free from flaws. Some researchers use this drawback to justify their claim that serious processes or protocols cannot be verified by this method [Arc02]. The IDS, the trusted platform and the environments in which they are embedded can only be modelled accurately by processes that have infinite states. Thus, we have to find appropriate ways to prune away details that are not required to spot policy violations. Additionally, we have to establish a technique to formally justify that these simplifications are not covering attacks. To do so we will use data independence techniques [Laz97]. The TCPA part will only use standard data independence techniques. The IDS analysis, however, requires more. We have to find a way to investigate the network topology, various data fields and the payload of the required communication protocols.

The third goal is to investigate the property of time in relation to intrusion detection. Usually processes or security relevant elements are validated without a notion of time, especially when the *trial–and–error* approach is used. In the history of various other fields, this has proven to be a fatal mistake. We suspect that time is not only a simple attribute in our intrusion detection models, but a complete dimension that can hide various vulnerabilities. We want to show that even in the case of a successful validation of our IDS model, by abstracting away

time, serious side-effects can remain undiscovered. To do so we will use two approaches: firstly, we will design an easy-to-build CSP model, and secondly, we will introduce a discrete time CSP model. The models themselves are kept as simple as possible. Finally, from this specific result, we will derive a method to reduce the complexity of a discrete timed process without losing attack possibilities, or more generally, without losing specification violations.

The fourth goal is to find a way to reduce the complexity of models, particularly those that evaluate internal hardware processes which act on external input provided by some security protocol. The internal transactions are usually complex enough to bring a complete state space exploration to it limits. In addition, the focus of such models lies not on the protocols themselves, but rather on the internal response of the addressed hardware component. Therefore, we have to find a way to reduce the external communication to only the necessary stimuli for exercising all possible behaviors (responses) of the hardware component, abstracting away from the details of the design of the protocols, and just modelling the services they provide.

1.4 Overview

Our general approach for analysing security mechanisms makes use of the process algebra CSP and its model checker FDR, as follows:

1. We model the honest participants in the security mechanisms using CSP;
2. We model the most general attacker who can interact with the security mechanisms;
3. We create a CSP specification of the security requirements;
4. We use FDR to explore the state space of the system, seeing whether or not it can reach an insecure state;
5. If FDR finds a reachable insecure state, then it returns the trace of actions that leads to that state, i.e. a successful attack.

Organisation Chapter 2 provides the reader with the relevant background information about Communicating Sequential Processes (CSP). CSP is a formal algebra that is used to model various communicating systems, such as Intrusion Detection Systems. The CSP part elaborates on the syntax and semantic models, more precisely on the traces, stable failures and the failures / divergences model of that calculus. Explaining the concept of data-independence, a technique to reduce the data complexity of the model, closes the CSP introduction.

In our introduction of intrusion detection systems, we classify them according to a very simple taxonomy. We distinguish between data source and detection method. Within these two classes we discuss the advantages and disadvantages of the most prominent subclasses, such as host versus network intrusion detection systems.

Chapters 3 – 5 represent the main part of the intrusion detection research in this thesis. In order to show that CSP is suitable to verify IDSs, we first consider the reproduction of known attacks. Once we reach that goal, we try to determine whether it is possible to apply the collected data to other areas of intrusion detection. To understand the first model, knowledge of IPv4 is required; therefore information on the basic functionality of this protocol is provided as well.

In section 3.3, we present a simple CSP model and show how FDR is used to detect flaws in that model. We consider whether the Internet Protocol version 4 (IPv4) [dR81] gives an attacker the opportunity to launch an undetected attack against the target. This model is based on a very simple packet structure, only consisting of the data and the Time-to-live field.

Section 3.4 describes a more complex CSP model. The packet structure of this model is greatly expanded now consisting of all fields relevant for packet reassembly. The algorithm that accomplishes the reassembling is based on RFC 791 [dR81].

The models thus far reveal attacks that were covered by [PN98]. In section 3.5 we reveal new attacks on heterogeneous networks, consisting of hosts that support different reassembly algorithms. We will construct a reassembly process based on RFC 815. However, we will only discuss the impact of weak specifications rather than suggesting any particular work-around, since there is no real work-around except that the RFCs should always be as unambiguous as possible.

In chapter 4 we discuss proof techniques for our models and the impact of feature abstraction on our survey. The disadvantage of using tools like FDR is that they explore every state of the model and therefore cannot deal with infinite state systems. As mentioned earlier solutions to this problem include applying abstractions on the general structure of the model, e.g. modelling fewer fields of IPv4, or restricting the scope of the modelled fields. However by applying these techniques it remains uncertain whether these restrictions do not cover other vulnerabilities. We show that detail we pruned away did not contain more attacks. We do so by showing how to generalise the network packet and attack signatures parameters for the time-to-live model. Additionally we show that precisely the same technique works for the packet reassembly model.

Afterwards we change the focus of the time-to-live model from Section 3.3, in order to move towards a more complete analysis, independent of the set of attack signatures. We show that a fairly restricted type for the TTL values is sufficient to capture all relevant behaviours of the system. We further show that a fairly limited buffer size for our reassembly buffer suffices. We also discuss why our

network topology is not as restricted as it appears to be and only hides *obvious* attacks.

In chapter 5 we inspect not only different ways to model time within the intrusion detection environment but we also derive from our very specific results tools for a more general case. More precisely, first we design an easy-to-build CSP model, and second we introduce an IDS model using discrete time. The models themselves are kept as simple as possible, consisting only of the required time-out mechanism that is used in IPv4 data transfers. However, we discuss them in more detail, since they are proof-of-concept designs. They should act as guidelines for more difficult problems. This examination closes with a discussion about how the two models are related to each other. From this we derive a function that converts every untimed timeout process using a sliding choice operator into a process that uses discrete time. The proof given shows that whenever the untimed model refines the specification then the discrete-time model does also. Thus giving us the option, for untimed safety specs, only to verify the process using the simpler timeout version without having to fear to miss a specification violation in the more complex discrete time model.

In chapter 7 we introduce the concept of trusted computing. We discuss current architectures proposed by Microsoft, the Trusted Computing Platform Association and Intel. The emphasis will lie on the TCPA solution. Moreover, we will describe scope, design features and various definitions of trust that are necessary for this architecture. This is followed by a more detailed overview of the different subsections of the TCPA model, such as integrity verification and reporting, creation of trusted identities and the protected storage. Since the architecture uses various protocols that have to be verified, we will also introduce *Casper* as a protocol verification tool that converts a protocol description language that is easy to write into machine readable CSP. Additionally, we will introduce common verification and abstraction techniques to evaluate the correctness of these protocols. The chapter finishes with a summary and a discussion about the relationships between the various trusted computing concepts.

Chapter 8 covers the authorization protocols. Every protected object on a trusted platform can only be accessed via an authorisation secret. TCPA defines five standard protocols that specify how one can generate, access and modify these secure objects. We will generate a CSP model of all five. More precisely, we will evaluate the Authorisation Data Injection Protocol (ADIP), the Object Specific - Authorisation Protocol (OS-AP), the Object Independent - Authorisation Protocol (OI-AP), the Authorisation Data Change Protocol (ADCP) and the Asymmetric Authorisation Change Protocol (AACP). Except for ADIP, all protocols are so called stream authentication protocols. Generally, they establish a shared secret and use this to encrypt an endless supply of nonces, consequently being able to generate a stream of authenticated commands. Therefore, special attention will be given to the abstractions that are applied to reduce the state space of these protocols.

Chapter 9 evaluates the context management of the Trusted Platform Module (TPM). The TCPA architecture [TCPA02, TCPA03d] tries to be as cost effective as possible. Consequently, it only demands that the TPM can store state information about two authorisation sessions. In practice, however, it is highly probable that the TPM has to deal with more parallel sessions. Hence, the TCPA introduces a caching model that allows the TPM to suspend sessions and externalise the state information. We generate a CSP model of all elements involved and verify whether or not the caching mechanism is secure. The problem with such a verification is that it not only involves the internal elements of the hardware component, but also authorisation protocol sessions. Hence, we use *Casper* to generate a CSP description of the authorisation protocol in use, and then add the internal transactions to the responder process (TPM). If one wants to evaluate a hardware component whose internal transactions depend on external input, this kind of scenario is very common. Therefore, we present a way to simplify these communications.

In chapter 10 we verify the boot sequence of a trusted platform. The CSP model we present includes an integrity-challenge response scenario to test whether or not the integrity reporting during the boot cycle can be used to vouch for the security of a platform. The term integrity-challenge response protocol is used by the TCPA to term the process in which a platform requests reliable information about another platform's system state. For this complete chapter we use a movie on demand system as an example set up. This chapter concludes with a discussion about the information value of the TCPA's integrity metrics and about various problems that could occur if one wants to rate them.

Chapter 11 discusses one usage scenario of TCPA — Digital Rights Management (DRM). Under the term DRM we understand: 'a system of information technology components and services that strive to distribute and control digital products' [Lyo01]. First, we will present issues and classification techniques of DRMs. Then we will discuss a protocol that could enable DRM; for this we will extend the AAC protocol. We will elaborate on the various advantages and disadvantages of our system.

Chapter 12 contains the conclusion of the thesis and a summary of the contributions made. It also presents related work and discusses how our own work relates to it. Finally, we present a forecast of avenues which we are interested to address in the near future.

Chapter 2

Background

This chapter provides the reader with necessary background information. We only introduce concepts that are essential for the proper understanding of the thesis. We proceed by giving an overview of the process calculus Communicating Sequential Processes. We will introduce its syntax and the semantic models used throughout this thesis. More precise we will elaborate on the traces, stable failures and the failures / divergences model. Since nearly all of our models use the traces model we will focus more on that model than on the others. The CSP overview is concluded by a brief introduction of data-independence. Further we will introduce Intrusion Detection Systems (IDSs) and classify them according to a very simple taxonomy that uses only data source and detection method as criteria. Within these two classes, we discuss the advantages and disadvantages of each member.

2.1 Communicating Sequential Processes

The motivation for using formal methods is that conventional ways of specifying systems rely heavily on natural language and diagrammatic methods. Such approaches make it harder to write unambiguous specifications and make it more difficult to analyse them. If omissions and errors introduced during the specification phase go undetected until late in the development cycle they become very expensive to rectify.

One well known formal method is Communicating Sequential Processes (CSP), invented by C.A.R. Hoare [Hoa78]. This version of CSP uses the concept of communication between processes by passing messages. This is currently the most widely used approach for inter-process communication (e.g., LAN, MAN and WAN). When Hoare published his first paper information was exchanged by handshaken communication. The book, Communicating Sequential Processes, the foundation of CSP version 2, was published in 1985 [Hoa85]. Since then vast amounts of theoretical work has been undertaken to improve CSP in order to

become more powerful and to cope with the new trials of the recent invention of concurrent programming [Ros98b]. We understand concurrency as the simultaneous execution of independent programmes / processes. These processes may or may not communicate with each other.

It is easy to imagine that to understand the behaviour of a system that consists of more than one process — processes that may be running at the same time — is much more difficult than mastering a system consisting of a single process. Unfortunately, because of the Internet and the fast growing use of distributed applications this is precisely what is required, especially if we want to understand the impact of certain environmental events on our system.

CSP is a language that describes processes. Additionally, it provides us with a great variety of semantic models to reason about the behaviour of processes. Since we do not assume any prior knowledge of CSP, we begin by introducing its notation. (For further information see, [Ros98b, Hoa85].)

2.1.1 Syntax

Events An event is a single, atomic, and instantaneously occurring action that a process engages in. There are two kinds of events; those that are external and those that are internal. The latter indicates that the system or process makes a decision without the environment. Such a decision can not be observed by the environment. The external event is chosen by the environment and is therefore observable.

Example 1 A cash change machine, is able to do two things – *take_cash* and *deliver_change*. A corresponding CSP process would look like:

$$CashChange1 = take\_cash \rightarrow deliver\_change \rightarrow CashChange1$$

This process is prepared to communicate the event *take_cash* and then engage in the event *deliver_change*. After performing these two actions it behaves like the process *CashChange1*, which means that it returns to its initial state.

The external choice Sometimes the defined process offers the environment the choice between performing two events. This choice is called external choice and represented by the symbol $\Box$.

Example 2 We could enhance our cash changing machine so that it is now possible for the user to decide between two different change possibilities.

$$\begin{aligned} CashChange2 = take\_cash \rightarrow (&deliver\_change\_one \rightarrow CashChange2 \\ \Box\ &deliver\_change\_two \rightarrow CashChange2) \end{aligned}$$

After the event *take_cash* takes place, this process gives the client the opportunity to decide between the events *deliver_change_one* and *deliver_change_two*. After that decision has taken place the process behaves like *CashChange2*; its initial state.

The non-deterministic choice The non-deterministic choice is similar to the external choice operator; the only difference is that the choice is taken internally, hidden from the environment. Therefore, the external system can not bias this choice. The symbol for internal choice is $\sqcap$.

Example 3 If we replace the external choice operator in Example 2 by internal choice, we get the following process:

$$CashChange3 = take\_cash \rightarrow (deliver\_change\_one \rightarrow CashChange3 \\ \sqcap deliver\_change\_two \rightarrow CashChange3)$$

In contrast to the process *CashChange2*, *CashChange3* does not give the user the opportunity to choose. The process makes the decision internally, hence we call such processes non-deterministic.

The hiding operator Sometimes it is necessary to hide events from the environment. This can be the case, for example, if we want to narrow our inspection to specific events. CSP has the hiding operator to manage this abstraction. For example, $P \setminus \{a, b\}$ stands for hiding all events a and b from the environment that are occurring in process P.

Example 4 Assume our *CashChange* process, is extended to introduce internal events such as sorting the received money (*sort_money*) or booking the delivered money in an account (*book_money*). However, these operations are not observable from the environment, therefore we make use of the hiding operator. The resulting process would be:

$$CashChange4 = take\_cash \rightarrow sort\_money \rightarrow \\ (deliver\_change\_one \rightarrow book\_money \rightarrow CashChange4 \\ \sqcap deliver\_change\_two \rightarrow book\_money \rightarrow CashChange4) \\ \setminus \{sort\_money, book\_money\}$$

The parallel composition There are four varieties of parallel compositions: synchronous parallel, alphabetized parallel, interleaving, and generalized parallel. We will give a brief description of each and how they relate to each other.

The synchronous parallel operator Using the synchronous parallel operator leads to a full synchronisation of all participating processes. This means that all participants must agree on all occurring events. It is represented by the symbol $\|$.

Example 5 If we want to model the interaction between the cash change machine and the customer we must also model the user. The customer is represented by:

$$Custom1 = take\_cash \rightarrow (deliver\_change\_one \rightarrow Custom1 \\ \Box\ deliver\_change\_two \rightarrow Custom1)$$

This is mainly the same process as *CashChange2*, because, as stated above, processes combined with the synchronous parallel operator have to agree on every event. The combined process is:

$$Combined = Custom1 \parallel CashChange2$$

If $Custom1$ or $CashChange2$ would contain an event that is not in the alphabet of its peer process then this event could never be activated.

The alphabetized parallel The latter constructor of parallel composition is very restrictive. Therefore, we have the alphabetized parallel operator. Its representation is $P1\ {}_X\|_Y\ P2$. This means that process $P1$ can communicate all events that are in set X and $P2$ can communicate all events in set Y. They freely communicate all events except those that are in the intersection of these sets. Once a process wants to engage in an event that lies within X and Y, both participants, in our case $P1$ and $P2$, have to synchronise upon it.

If X and Y are equal, there is no difference between the alphabetized parallel and the synchronized parallel. Conversely, where the intersection of the sets is empty this operator behaves like the interleaving operator introduced below.

Example 6 In Example 5, our customer process had to offer the same events as process *CashChange2*. This does not make sense, especially the *take_cash* event. Therefore we change our customer process to become:

$$Custom2 = donate\_cash \rightarrow \\ (deliver\_change\_one \rightarrow take\_change\_one \rightarrow Custom2 \\ \Box\ deliver\_change\_two \rightarrow take\_change\_two \rightarrow Custom2)$$

The complete system would then be:

$$Combined2 = Custom2 \ {}_X\|_Y \ CashChange2$$
$$where \ X = \{donate\_cash, deliver\_change\_one, deliver\_change\_two,$$
$$take\_change\_one, take\_change\_two\}$$
$$\wedge \ Y = \{take\_cash, deliver\_change\_one, deliver\_change\_two,$$
$$take\_change\_one, take\_change\_two, sort\_money, book\_money\}$$

This gives the customer process the freedom to engage in the event *donate_cash* without synchronising with *CashChange2*.

The interleaving operator We need the interleaving operator, represented by $|||$, to model two processes that are connected but acting independently from each other. Therefore, in the combination $P1 \ ||| \ P2$, all events in $P1$ and all events in $P2$ can occur completely independently. If an event occurs that both processes could have communicated, then the choice of which one executed it is non-deterministic.

Example 7 Suppose we want to model a system that consists of two processes that are not related at all. One example could be the ATM and the cash changing machine at a bank. The resulting process would be:

$$BankSystem = ATM \ ||| \ CashChange2$$

The generalized parallel operator All three of the above operators are combined in one operator called, generalized parallel. The notation for this operator is, $\underset{Y}{\|}$. $P1$ and $P2$ have to agree on all events that are in set Y. Hence, we derive the interleaving operator from the generalized parallel operator by making Y the empty set. The alphabetised parallel is modelled by satisfying the following condition: Y is the intersection of the alphabet of processes $P1$ and $P2$.

The channel notation A channel is a compound object, where an infix dot functions as a delimiter. For instance, a channel a with parameter over some type T is written as $a.T$. This again is equal to $\{a.x | x \in T\}$ of our alphabet (Σ). The channel definition itself does not specify whether the values of type T are inputs or outputs over a. The process itself has to define this by using the symbols ! and ? for output and input respectively (e.g. $a?x : T \rightarrow P'(x)$). A channel can combine as many objects as required.

Example 8 If we want a model of the money slots of the cash changing machine as well, we can make use of channels.

$$Slot = donate\_cash?x : Money \rightarrow take\_cash!x \rightarrow Slot$$

This process works as an interface between our $CashChange$ processes and the customer process. It takes a value of type $Money$ on channel $donate\_cash$ and binds it to variable x. It then submits the value stored in variable x on channel $take\_cash$ to the $CashChange$ process.

The renaming operator Sometimes it is required to map certain events or channels to other events or channels. This can be achieved by the renaming operator: the process P$[|a \setminus b|]$ behaves exactly as P except that the event or channel b is replaced by a.

Example 9 Assume we have the similar situation as in example 7, the only difference, that now we combine two cash changing machines. Therefore, we get:

$$BankSystem1 = CashChange2 \;|||\; CashChange2$$

However this would lead to a serious problem: if we assume that each machine is used by one customer, we have no criteria to decide between the $deliver\_change\_one$ and $deliver\_change\_two$ events of these machines. To distinguish between these two machines we use the renaming operator, which leads to the following system:

$$\begin{aligned} &BankSystem2 = CashChange2 \;|||\\ &\quad (CashChange2\; [|take\_cash2 deliver\_change2\_one, deliver\_change2\_two\\ &\quad \setminus take\_cash, deliver\_change\_one, deliver\_change\_two|]) \end{aligned}$$

Using the resulting process *BankSystem2*, we can now distinguish between the two machines.

Modelling timed behaviour The standard CSP models only consider the order of events without the notion of time. There are two approaches how one can model time in CSP: Timed CSP [Sch00a, Ros98b] and un-timed CSP [Ros98b]. Timed CSP can be divided further into continous and discrete timed CSP [Sch00a, Oua01]. The first of these introduces time by pairing every event in a trace with a corresponding timestamp: a trace looks like $\langle (timestamp, event) \rangle^\frown t$ rather than $\langle event \rangle^\frown t$. It uses a dense or continuous model of time. The timestamp is a non-negative real number that will be increased until the event happens; thus, it has an infinite state space. This prevents one from using tools like FDR. [Ros98b] mentions that the only way this infinity can be counterbalanced is by 'imposing severe restrictions and using clever equivalences'. The discrete approach remedies this drawback. The time is projected onto an infinite number of discrete instances. These instances are represented by the special event *tock*. This event can stand for any amount of time passed. [Oua01] discusses the relationship between discrete and continous timed concurrent systems. An approach similar to the discrete

timed CSP, is discussed in [Ros98b]. It uses an event called *tock* to represent the passage of time. However, this approach facilitates only untimed CSP operators and the event *tock* is not treated differently from any other event. Every process in the modelled system has to synchronise with the other participants on *tock*. Otherwise, we would not be able to relate the time passed in the overall system to the time passed within each process.

Another way to simulate timing behaviour (timeout) is to use the sliding choice (also called the timeout operator $\triangleright$). It is originally derived from $(a \rightarrow P \ \Box \ b \rightarrow Q) \setminus b$, which is equivalent to $(P \sqcap STOP) \ \Box \ Q$. However, the resulting traces are far from obvious – especially in complex models. In chapter 5 we will present two approaches to model the timeout behaviour of the TCP/IP stack. The first design does not use time, instead it uses the sliding choice operator to achieve the same behaviour as an IPv4 timeout mechanism. The second model uses a *tock* based timeout.

2.1.2 Semantic models

The traces model A trace of a process is a sequence of visible events that the process can perform. The traces of a certain process is a set that contains all finite traces that this process can generate. The traces model describes the processes in terms of possible traces that a process can perform. As described in [Ros98b] the function $traces()$ delivers for each process the set of all its finite traces. For example, once we apply the function $traces()$ to the process P, where P is defined as $P = e \rightarrow P$, we get $\{\langle e \rangle^n \mid n \in \mathbb{N}\}$. Since we are using the traces for proofs in the later chapters it is worth looking at some properties of this set as well as how the CSP operators are behaving in this model. The following two properties are always true if we apply $traces$ onto P, whereby P stands for any process:

1. $traces(P)$ has at least one element; it always contains the empty trace.
2. $traces(P)$ is prefix closed: whenever $s^\frown t$ is an element of $traces(P)$ then s is also.

It is possible to use a few rules to calculate this set by hand. In our case this is not important since we use a model checker called FDR to do this task for us. However, as mentioned before, to prove that our abstractions, that we will establish during our examination are sound we will require the following rules:

1. The resulting set of $traces(a?x : A \rightarrow P)$ contains the empty trace the initial event $a.y$, where the y is an element of the set A, followed by a trace of process P. This results in $\{\langle\rangle\} \cup \{\langle a.y \rangle^\frown s | y \in A \wedge s \in traces(P[y/x])\}$. The notation $P[y/x]$ expresses that all free occurrences of the variable x have the value y.

2. The resulting set of $traces(P \ \Box \ Q)$ equals $traces(P) \cup traces(Q)$, since this process can behave like P or like Q, depending on the decision of the environment.

3. $traces(P \sqcap Q)$ provides us with the same result as $\Box$. Hence, in the traces model we can not distinguish between internal and external choice.

4. For $traces(\text{if } \mathit{bool} \text{ then } P \text{ else } Q)$ we obtain, if the expression *bool* holds, the result $traces(P)$ otherwise we get $traces(Q)$. Overall we obtain $traces(P) \wedge traces(Q)$.

5. $traces(P \ || \ Q)$ as mentioned earlier P and Q have to agree on all events they engage in therefore we get $traces(P) \cap traces(Q)$

6. $traces(P_X \|_Y Q)$, since process P and Q have to agree on communicating events in the intersection of X and Y, this equals $\{s \in (X \cup Y)^* | s \upharpoonright X \in traces(P) \wedge s \upharpoonright Y \in traces(Q)\}$

7. $traces(P \ ||| \ Q)$ equals $\bigcup\{s \bigotimes t | s \in traces(P) \frown t \in traces(Q)\}$ where the operator $\bigotimes$ is defined as follows:

$$\begin{aligned} \langle\rangle \bigotimes s &= \langle s \rangle \\ s \bigotimes \langle\rangle &= \langle s \rangle \\ \langle a \rangle \frown s \bigotimes \langle b \rangle \frown t &= \{\langle a \rangle \frown u | u \in s \bigotimes \langle b \rangle \frown t\} \\ &\quad \cup \{\langle b \rangle \frown u | u \in \langle a \rangle \frown s \bigotimes t\} \end{aligned}$$

8. $traces(P \setminus X) = \{s \setminus X | s \in traces(P)\}$.

Refinement With the refinement operator we can express a superiority or inferiority relation between two processes. The refinement follows one simple equation,

$$R \ \sqsubseteq \ P \ \equiv \ R \ = \ R \ \sqcap \ P$$

which means that P refines R; or we can say, P is more deterministic then R. In terms of CSP we then say that the process P is better then R. Since the set of traces of P is a subset of the set of traces of R, therefore whenever R satisfies a given specification then P does also. This relation can be expressed by the symbol $\sqsubseteq$. Using the traces model we get:

$$R \ \sqsubseteq_T \ P \ \Leftrightarrow \ traces(R) \ \supseteq \ traces(P)$$

Refinement has a few interesting properties, such as transitivity.

$$P \ \sqsubseteq_T \ Q \ \wedge \ Q \ \sqsubseteq_T \ R \ \Rightarrow \ P \ \sqsubseteq_T \ R$$

This means that, whenever process P is refined by process Q, and Q is refined by R then P is refined by R. As we will require later, all CSP operators are

monotonic with respect to refinement. If we use the process context $F[\cdot]$ to alter a process called *System* and the result refines a certain specification called *Spec*. We can split the proof that,

$$Spec \sqsubseteq_T F[System]$$

into two parts, by using an intermediate process called P. If following statement holds,

$$Spec \sqsubseteq_T F[P] \wedge P \sqsubseteq_T System$$

then by monotonicity:

$$F[P] \sqsubseteq_T F[System]$$

Finally we have to use the transitivity of the refinement operator to establish our initial statement:

$$\begin{array}{c} Spec \sqsubseteq_T F[P] \wedge F[P] \sqsubseteq_T F[System] \\ \Rightarrow Spec \sqsubseteq_T F[System] \end{array}$$

The stable failures model The *traces* model describes processes in terms of their possible behaviour. However, this is sometimes not enough, because it can only capture safety properties. Additional information about what the process, after performing a certain trace tr, refuses to do would be desirable. As an example the operators $\sqcap$ and $\Box$ are indistinguishable by just inspecting the traces. The *stable failures* model extends this model to capture liveness properties.

A failure of a process P is a pair (tr, X), representing that the process can perform trace tr to reach a stable state, where no event of the set X can be performed. A stable state is a state where no internal activity is possible. We only consider stable states, since by definition of a failure the refusal set of a certain state should be refused regardless of how long its events are offered. A process Q failure refines another process P, written $P \sqsubseteq_F Q$ iff:

$$traces(P) \supseteq traces(Q) \wedge failures(P) \supseteq failures(Q).$$

The failures/divergences model The *stable failures* model can not handle divergence. A diverging process such as $P = a \rightarrow P \setminus a$ is not performing usefully nor refusing anything. The *failures/divergences* model relieves us from such shortcomings. Because we cannot trust process P, once it has reached a state where it can diverge, to do anything, we can conclude that two processes that can diverge immediately are not only equivalent but also completely useless. Once a state is reached where the process can diverge we assume that it can

engage in any action, thus creating any trace, refuse Σ, and diverge in any later stage. Thus, $divergences(P)$ provides us with all traces P can diverge from and their possible extensions. This changes the sets $traces$ and $failures$. The, now called, *strict sets* deliver the following elements:

$$traces_{\perp}(P) = traces(P) \cup divergences(P)$$
$$failures_{\perp}(P) = failures(P) \cup \{(s, X) | s \in divergences(P)\}$$

A process P in this model is represented by:

$$(failures_{\perp}(P), divergences(P))$$

The refinement relationship $P \sqsubseteq_{FD} Q$ holds iff

$$failures_{\perp}(P) \supseteq failures_{\perp}(Q) \wedge divergences(P) \supseteq divergences(Q)$$

Failures/divergences refinement In the early days CSP was just applicable to simplistic models because the analysis had to be done by hand. However, today there are various tools that support this technique. One possible tool is Failures/Divergences Refinement (FDR). FDR is fed by a slightly modified notation of CSP called CSP_M (machine readable CSP). This tool parses the CSP_M code and creates a state transition graph to check whether there is a possible trace leading to a deprecated state. FDR is maintained by Formal Systems (Europe) Ltd.. We will use FDR for verification of models presented in this thesis [Ros98b, GGH+00].

2.1.3 Data-independence

Usually, the behaviour of CSP processes is dependent upon various parameters. These parameters are used to enable different instantiations for the processes. The problem, in combination with FDR, FDR explores only one instance at a time and its search is exhaustive. Hence, the parameter range can only be small. The underlying problem is called the Parameterized Verification Problem (PVP). It tries to address whether the specification holds for all instances [Ros98b, RB99, Laz97].

Data independence is a tool that can be used to address this problem. The goal of this approach is to come up with a bound N such that if a refinement holds whenever a type parameter is instantiated with a type of size N, then the refinement also holds for all larger types. The core of this is summarized by following statement.

> A process P is said to be data-independent with respect to a type T if the only operations it performs on values of T are to input them, store them, and output them, but never perform any 'interesting' computations on them that constrain what T might be[Ros98b].

A type T is data-independent with respect to a programme P precisely when P satisfies all the following conditions:

1. Values of T do not appear in the source code of P.
2. The only operations that are allowed in context of T are that passing of elements of T around without looking inside them.
3. No predicates should use elements of T. The exceptions are equality testings, implicit (e.g., synchronization on a value of T) or explicit (e.g., If Then Else).
4. Operations that extract information about data type T may not appear.
5. The programme should not contain any replicated constructs that are indexed by T, except the nondeterministic choice.

As stated in the introduction of this section, data-independence is about calculating a certain threshold[1] N, where $N \leq M$ and M represents the number of elements of type T, both N and M are natural numbers. Therefore, if $Spec(T) \sqsubseteq_T P(T)$ and T is restricted to N values holds then the refinement holds for the full range of T. The remaining problem is the calculation of the threshold.

For this we have to introduce two conditions: No Equality Testing (NoEqT) and Norm.

The No Equality Testing condition (NoEqT) This condition holds for P when it does not contain explicit or implicit equality testing regarding elements of T. By explicit, we mean $if - then - else$ statements and by implicit, synchronizations between processes, on elements of T.

Theorem 1 *If NoEqT and data-independence holds for both the specification and the implementation, and the Specification does not restrict the Implementation regarding T, the threshold is one [Ros98b].*

The remaining part is to define when *Spec* meets the Norm condition. This condition is met if we abide by the following rules:

1. it does not contain the hiding nor the renaming operator;
2. it does not contain parallel operators, except the alphabetized parallel and its replicated version;

[1]The state that is reached by using the threshold as data size for the data type is often called data saturation [Ip96, GJ88, ID93]. This expression originates from the fact, that even if we increase the size of T, the system's offered behaviour essentially remains the same.

3. no replicated nondeterministic choice operator is allowed that's index is built upon T;

4. in order to know which side of the choice was taken, all internal and external choice operators and the time-out operator must have the set of initial events of each argument disjoint from each other;

5. the sequential composition and interrupt operator should be used in such a way that the former point is not violated implicitly; for the exact technical conditions, see [Laz97].

Hence a process satisfies Norm_T if, essentially, it contains no nondeterminism the effects of which are not immediately apparent (see [Laz97, Ros98b] for a formal definition).

As we will see, in one of the presented cases the specification restricts the implementation after a given trace, in communicating certain elements of T. Therefore we have to use another of Lazic's theorems. The following theorem is taken from [Laz97, Ros98b].

Theorem 2 *Suppose Spec and Impl are data-independent processes, both satisfying NoEqT and Spec satisfying Norm. Let* $\sqsubseteq$ *be any of* $\{\sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}\}$.

1. *If* $Spec(2) \sqsubseteq Impl(2)$ *holds (i.e. for T of size 2) then* $Spec(m) \sqsubseteq Impl(m)$ *holds for all finite and infinite* $m \geq 2$.

2. *If the refinement* $Spec(2) \sqsubseteq Impl(2)$ *fails then* $Spec(m) \sqsubseteq Impl(m)$ *fails for all finite and infinite* $m \geq 2$.

This theorem will be used in Chapter 4 to restrict the unbounded supply of different network messages. The data-independence framework offers many more theorems, the interested reader is referred to [RB99, Laz97, Ros98b]. Since we mainly focus on data independence we will only give a brief outline of another promising abstraction technique.

Predicate abstraction is another abstraction technique that is mainly used in the realm of software verification [BaRa, GS97]. Predicate abstraction first appeared in [GS97], since then it has been the focus of various other research projects [FQ02, Rob].

In predicate-based abstraction methods, the data in the real world system is abstracted by only keeping track of certain predicates that are based on the data. Every predicate is represented by Boolean variables in the abstract program, while the original data variables are pruned away. Early applications of this method (for example [GS97])were dependent on the user identifying the set of predicates that influence the control flow. However more recent research projects

describe algorithms that compute relevant predicates and an abstract program based purely on a syntax-directed analysis of the corresponding program text.

Data independence on the other hand focuses on the abstraction of data types of variables that do not influence (or only in a very restricted way) the program control flow. Thus, we could never use data independence to abstract systems where elements of the data type in question are used for complex arithmetic operations and the results are used to decide the succeeding behaviour; however we could use predicate abstraction. In contrast to this, a system that uses variables of a data type with infinitely many elements and all these variables are not bound to decisions and evaluations within the system, could not be abstracted by predicate abstraction — but by data independence. We will not further pursue predicate abstraction the interested reader is referred to [GS97].

The overview of CSP is far from complete however the material that will be used to reach our goals was covered. The next area we have to introduce before starting our analysis concerns intrusion detection within computer networks.

2.2 Intrusion detection systems

An Intrusion Detection System (IDS) is a system that detects abuses, misuses, and unauthorised uses in a network. The great advantage of an IDS is that it can spot security breaches from insiders as well as outsiders. These systems identify intrusions by spotting known patterns or by revealing anomalous behaviour of protected resources (for example, network traffic or main memory usage) [HSTL90, Ilg93, Lun93].

This line of research was started of by Anderson [And80] in 1980. Since then it has been an active field of research. In the beginning the progress and the awareness for the need of such systems was low. However, in 1987 Denning's proposal [Den87] of a framework for intrusion detection systems initiated a rally in this field, even now many commercial products use her framework.

The following section contains all relevant information about IDSs to understand this thesis. We give a small description of the different classes of IDS that are currently available in industry and academia. We will also discuss the common advantages as well as the drawbacks that the systems of a particular class have.

Classification Criteria We distinguish IDSs by their detection principles and by their different kinds of raw event sources. At this point we will omit a classification according to the system structure. However, we will mention this point in the conclusion. Further information on classifying IDSs is given in [Axe00].

2.2.1 Detection principles

Detection principles signify the method of detecting whether or not an attack is in progress. This can be regarded as a specific instance of the more general signal-noise detection problem [Ega75], whereby the manifestations of attacks represent the signals and the background noise is represented by all operations that do not violate the security policy. The general difficulty is to establish a correlation between an observed operation and the signal or noise distribution.

Physics solves this problem with the knowledge of both distributions. However, current IDSs only use one distribution to fulfill this assignment. They use either the signal distribution, in IDS terms represented by signature-based detection mechanisms, or the noise distribution, the anomaly-based detection approach.

We can split these two classes of IDSs into classical signature detection, negative signature detection on one side and into policy based detection and anomaly detection on the other. The last detection principle, called hybrid detection, independently uses a signature and an anomaly based detection engine.

Misuse detection Misuse detection based systems look for known signatures of attacks. They are also called signature detection based systems. A *signature* is the pattern that is used by the IDS to spot attacks [Ken01, Sun96]. It is a specific manifestation of a certain attack. But not only systems that use pattern matching to examine a specific data source fall into this category. [Kum95], for instance, uses state machine descriptions to spot attacks. [Ilg93] uses a state transition analysis technique to reproduce the state of an observed object. The signatures that are necessary for these systems are mostly developed by hand.

The IDS usually obtains the required information from a network adaptor, which feeds it with raw data packets, or from the log-files of the hosting operating system. In industry the most used systems are network signature based IDSs [MB01, Nor99], because of their low total cost of ownership. More examples and a detailed description of these systems can be found in [Axe00, Pax99, Sec, SNO].

Advantages The system knows exactly how a certain attack manifests itself. This leads to a low false-positive ratio. The detection algorithm is based on pattern matching, for which efficient solutions exist.

Disadvantages Defining the manifestations of certain attacks is a time consuming and difficult task. Due to the working principle of these systems, it is nearly impossible for them to detect novel attacks. Moreover, subtle variations in the attack can mislead them.

Negative Signature Detection Negative Signature Detection was originally introduced to overcome the sinister side of signature detection. It should, therefore, be able to detect novel attacks. To address this task a negative data set must be employed. In other words, the system administrator defines signatures / datasets of normal operations. If the monitored event deviates from the data set, an alarm will be raised. This kind of IDS exists only in academia. It is only practical if we have a very restricted set of legal operations otherwise this would result in a high false-positive ratio. Therefore, we will not describe it in detail.

Specification-based detection [KFL94, Ko96] and [SBS99] were some of the first papers that recommended this approach. They distinguished between normal and intrusive behaviour by monitoring the traces of system calls of the target processes. A specification that models the desired behaviour of a process tells the IDS whether the actual observed trace is part of an attack or not. With this approach, the attempt was to combine the advantages of misuse and anomaly detection. It should reach the accuracy of a misuse detection system and have the ability to deal with future attacks of anomaly detection. These systems manage the detection by inspecting log files. This differs from [SU], where a run

time engine was developed to detect violations in real time. This approach is also capable of intercepting intrusions.

Advantages The advantages of these systems are fundamentally similar to those of the misuse detection systems. However these systems manage to detect some types/classes of novel attacks. Finally, they are more resistant against subtle changes of attacks.

Disadvantages Usually for every program that is monitored, a specification has to be designed. Furthermore, the modelling process can be regarded as more difficult than the design of patterns for misuse detection systems. Additionally some classes of attacks are not detectable at all.

Anomaly detection Such systems distinguish between normal and anomalous behaviour of guarded resources ([SRI02, SRI01]). Examples of monitored resource characteristics include CPU utilisation, system call traces, and network links. The decision regarding which behaviour class currently relates to certain events is made by means of a set of profiles. The profiles of normal behaviour for a resource are maintained by a self-learning algorithm. The characterisation of the normal behaviour is nontrivial. [Lun90] uses a statistical approach to model the system's behaviour, whereas [HSTL90] uses predictive pattern generation. More recent inventions are using neural networks [Lun93] or try to use the immune system as a role model for the perfect IDS [HF00, KB01].

Advantages The cost of maintaining the system is usually low, because the profiles are updated by the self-learning algorithm. Additionally, it can detect novel attacks as well as variations of already known ones.

Disadvantages The self-maintaining algorithm is usually computationally expensive. Sometimes unusual behaviour is not a precise indicator of an ongoing intrusion. [Bel93] discusses the fact that '[...]it is very usual to see unusual TCP/IP traffic[...]'. The result is a high false-positive ratio [JM00]. Finally, these systems can learn to classify intrusive event traces that are performed slowly as normal behaviour, which renders them useless.

Hybrid Detection Hybrid Detection systems combine the two approaches of signature and anomaly detection [LA00]. Academics and commercial vendors are trying to overcome the drawbacks of signature based systems — their reduced scope and inability to detect new attacks. They are also trying to overcome the problems of anomaly based systems — their large false positive rate — by combining them with signature detection. If we look at currently deployed systems, we notice that 80 to 90 percent are signature based while only 10 to 20 percent

of the detection capability is anomaly based. However, this latter proportion is likely to increase since several promising research projects into anomaly based IDS have been launched.

2.2.2 Different kinds of raw event sources

Another possibility for classifying IDSs is by means of the place and methods by which they collect the information. Three different types can be distinguished — a Network based IDS (NIDS), a Host based IDS (HIDS) and a Stack based IDS (SIDS).

Network Intrusion Detection Systems (NIDSs) An NIDS gets its information from a network adapter operating in promiscuous mode. It examines the traffic for an attack symptomatic signature [SNO, Sec, ISS05, Cis04]. Although anomaly detection has been implemented for these systems [SNO, Cis04], the main detection principle remains the misuse detection. A NIDS can provide surveillance for a whole network, because it is working with the raw network packets. In our further examination we model an NIDS, because it is the most used system type [MB01].

Advantages Due to the fact that a single NIDS can monitor a whole network its implementation and maintenance costs are low. Additionally, since they work at the packet level, these systems have all the information to sift out the difference between hostile intentions and friendly intentions. After a successful break in, the attacker usually wants to erase his footprints, thus deleting the audit logs of the host and its Host Intrusion Detection System (HIDS). In the case of using an NIDS, all activity is logged by a different system; this makes it difficult to delete them. The speed of these systems is another advantage - it detects attacks as they occur in the network. This gives it the opportunity to react before serious damage is caused.

Disadvantages These systems are largely unable to read the traffic of encrypted connections. The only exception would be to include them into the security association [HK98, BW97]; however this can be regarded as computationally too expensive. Nevertheless, users have increasingly encrypted their communication, rendering the system obsolete. Additionally, more and more networks are switched rather then broadcasted. Due to the Ethernet working principle [Bla98], the NIDS is only able to collect packets that travel through its collision domain. In a switched environment, there is no real collision domain; hence the NIDS is not able to retrieve vital information [Sys99]. One possibility is to use the mirror ports (SPAN) on the switch to collect all packets. However, such a SPAN port can easily run out of capacity [Sys99].

Host Intrusion Detection Systems (HIDSs) The HIDS runs on a specific host and watches its logging activity [Sys00, Nor99, Tan01]. Therefore, these systems are operating system dependent and every protected host needs a separate IDS [Sys98]. They can keep track of all actions that are made by the users of that host which include browsing for files with the wrong read/write permissions, the adding and deleting of accounts, and the opening and closing of specific files. This gives them a great aptitude for surveillance of security policy violations. Further examples can be found in [AXE98a, AXE98b, Ein01, Els00, Inn01].

Advantages The system knows whether or not an attack is successful. It usually produces a reduced number of false-alarms caused by unsuccessful attacks; for example, a HIDS protecting a Linux host would not raise an alarm if an attacker sends a Microsoft IIS Buffer Overflow Attack against this host [CER02]. HIDSs have more monitoring variables than NIDSs. Because HIDSs reside on the target host, these systems are able to keep track of encrypted end-to-end connections. Some of them even watch the packets as they traverse up the TCP/IP stack, allowing them to drop a packet that would lead to a security policy violation [LA00]. Another advantage of this structure is that HIDS do not require additional hardware.

Disadvantages For every monitored node a HIDS is required. This makes them very expensive in maintenance. HIDS are operating system dependent, therefore different implementations for different operating systems are required which makes them expensive in development. They also reduce the operational capacity of a network node, because they run their analysing processes in parallel with the business applications. Additionally, once an attack succeeds one has to trust information collected from a corrupted host. Ranaum agues that: *It is inherently flawed to trust data of a corrupted host* [Ran01]. Finally they have serious problems to detect more elaborate attacks; involving more network notes (i.e. distributed host scans).

Future directions in intrusion detection To discuss the future development of IDSs we have to distinguish between the progress made in industry and that achieved in academia. Since the corporate networks are getting more sophisticated, in particular, larger, more distributed and with more entry points to other, potentially hostile, environments, industry focuses on data collection, correlation and presentation [CER03, JH03]. Internet Security Systems, for instance, uses micro agents, slim and efficient data collection engines, that are deployable on every host [Sys00b, BR]. This clearly leads to an information flow problem. If the data is subsequently evaluated, one has to answer the question of how and where the data will be processed and stored. The data can be processed and stored completely on site. It can be processed partially by the agent; in this case,

however, the data is stored at a central point that executes advanced correlation techniques. The third option would be that every agent's purpose is pure data extraction and the evaluation is completely undertaken by a central management station [AXE98a]. Clearly an evaluation on site represents the most efficient way in respect of network utilisation. However, for complicated analysis, this may reduce the operational capacity of the host. Additionally, it prevents the IDS from detecting elaborate attacks such as distributed attacks [Nor99]. The approach where the central management station is performing all the attack detection, results in a heavy utilisation of the network capacity. Furthermore the node that works as a central management station can easily run out of resources. Hence elaborate load balancing techniques have to be applied, which clearly complicate the whole system [Sys99]. The remaining approach where some processing is done on site and the evaluation of distributed attacks performed by the central detection engine seems to be the favoured approach.

Since 'data is not information, information is not knowledge and knowledge is not wisdom' industry tries to address data presentation and correlation as well. [CER03, JH03] present two approaches to provide the administrator with comfortable GUIs. For data correlation, industry pursues various approaches. Most of them are based on pattern-matching or similar techniques that show an advantageous run-time complexity. However, the most stimulating work in this area is done in academia.

Clearly all the issues addressed above are important for academia as well. For instance [LS98, LNY+00] are dealing with data correlation. [LS98] uses data-mining algorithms to generate profiles for anomaly IDSs. More precisely, they use the association rules algorithm and the frequent episodes algorithm to learn the intra- and inter-audit record patterns to embody the behaviour of certain processes. [LNY+00] adds another dimension to this approach. They use the Common Intrusion Detection Framework (CIDF) [CIDF] to develop a prototype of a distributed IDS that consists of multiple agents that can independently detect hostile patterns. These agents continuously update their data-mining resources. After applying the mining algorithms, this allows them to learn representations of novel attacks. These newly learnt patterns can then be communicated to other agents. This research project covers not only correlation and topological issues, but also the interoperability between other IDSs — which will become increasingly important. In a similar vein, [MRS01] uses clustering to associate legitimate users with certain profiles. They then simplify the collected data by applying a genetic algorithm. The process finishes with the refinement of the associations between users and profiles by using a neural network.

Another field within the intrusion detection area is represented by the underlying attack classification and representation theory [How97, Kum95, Krs98]. For example [How97] developed a taxonomy for describing and classifying attacks. He founds his broad categorisation mainly on the dimensions attackers, tools, results and objectives. [Kum95], on the other hand, uses pattern specifi-

cations and refinement, based on representability, to create a hierarchy between different categories of intrusions.

2.3 Summary

In this chapter, we introduced the necessary background information for the remainder of this thesis.

We gave an overview of Communicating Sequential Processes in some detail. We elaborated on the syntax and semantic models that can be used to reason about various attributes of processes. The focus of our introduction was on the CSP operators and the traces model. Additionally we gave a brief introduction of the stable failures and the failures / divergences model. The CSP overview concluded with an introduction of the data-independence technique. This technique enables the reduction of certain infinite state systems to finite state systems without loosing relevant detail.

In the second part of the introduction, we established a simple taxonomy to distinguish between different types of intrusion detection systems. This taxonomy uses the attributes data source and detection method to elaborate on the differences between anomaly, misuse and specification based detection on one side and NIDSs and HIDSs on the other side. We discussed their advantages as well as their disadvantages. The intrusion detection part finishes with a outline of future research in that area.

Chapter 3

Intrusion detection systems CSP models

In order to show that CSP is suitable to verify IDS we first try to find efficient ways to reveal already known attacks. Once we are able to do so, we look to find whether it is possible to use our technique in new areas of intrusion detection. The remainder of this chapter is organised as follows:

To understand our models it is necessary to have basic knowledge of IPv4. Therefore, in section 3.1, we will provide the relevant background information.

Section 3.2 is setting up the stage by discussing the abstractions and assumptions that are required to keep the state space low.

In section 3.3 we investigate whether the Internet Protocol version 4 (IPv4) [dR81] gives us the opportunity to encode a well known attack against a monitored target in such a way that our IDS can not spot the threat. The model is based on a very simple packet structure, only consisting of the data and the Time-to-live field and should show how FDR is used to detect flaws in that model. This section finishes by discussing a suggested work-around.

Section 3.4 describes an enhanced CSP model, whereas its simulated packet structure will cover more fields. The packet consists of all relevant information to allow a proper packet reassembly. The reassembly algorithm itself is purely based on RFC 791 [dR81]. Finally prevention techniques are suggested.

In section 3.5 we build a new model upon the knowledge gathered from our first two models. We will construct a non-deterministic process based on RFC 815. The non-determinism covers all reasonable choices a programmer had to make if he wants to implement a RFC 815 compliant reassembly algorithm. The choices purely reflect the leeways given by the ambiguous specification [Cla82]. The modelled network consists of nodes that use different reassembly algorithms. The IDS for instance uses RFC 815 as guideline and the target reassembles according to RFC 791; considering the heterogeneous environment in today's networks this is a usual setting.

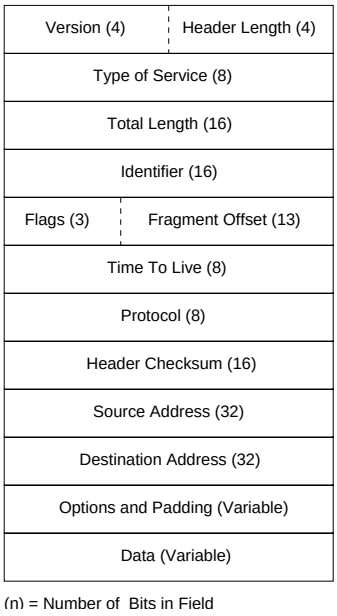

Figure 3.1: IPv4 header

3.1 Internet protocol version 4

As mentioned above, this analysis not only focuses on modelling IDS with CSP, but also on modelling the environment of an IDS. The first crucial component to understand is the Internet Protocol version 4 or IPv4. This section introduces this protocol. However, we will only discuss the fields and functions that are relevant for our first security inspection. (For further information see [dR81].)

Figure 3.1 presents the structure of an IPv4 header based on RFC 791. The followings a short description of each field.

The Version field is 4 bits long and identifies the version of the IP packet. Currently this value is always four. However, the IPv6 protocol has already been released [HD98b] and will be introduced in the near future.

The Identifier field becomes important if the datagram is fragmented. In this case the receiving host has to distinguish between fragments that may belong to different packets. A host uniquely identifies what fragment belongs to a certain packet by evaluating this field and the address fields.

The Flags field consists of three bits that are required for the fragmentation algorithm:

1. Bit 0 is reserved and is therefore not in use.

2. Bit 1: a one assigned to this position indicates that the IP packet has to stay un-fragmented, whereas a zero indicates that fragmentation is permitted.

3. Bit 2: a one in this position indicates that more fragments are to follow, whereas a zero indicates that this is the concluding fragment.

The Fragment Offset value describes where this fragment belongs within the original IP packet.

The Time to Live field maintains the distance a packet can travel: every router decreases its value by one; once zero is reached the packet is discarded.

The Source Address indicates where the packet originated from.

The Destination Address identifies the target of the packet.

The Data field accommodates the user data.

3.2 Modelling assumptions

The CSP models are built under certain assumptions. One general assumption is that the IDS is an NIDS based on signature detection. We believe that this is the most relevant IDS because of their widespread use. The IDS itself is considered to be perfect, in the sense that it knows all vulnerabilities that could be used to cause a security breach. We now consider only one-way and in-order communication. Further, we assume that a channel or device cannot corrupt a packet, nor can it refuse to forward a packet when it ought to do so (the only exception is the timeout model).

We now consider the network topology. We model a network with just one sender and one receiver node. We use a DeMilitarised Zone (DMZ) configuration, which is commonly used in industry [CZC00]. It consists of an exterior filtering router and an internal filtering router (see Figure 3.2 below); the exterior one is responsible for protecting the network from most attacks; the interior one is the most restrictive, as it only allows traffic that is permitted for the internal network. The DMZ resides between these two routers; this is the place where companies maintain their public servers, such as the web server. This is also the preferred place for the IDS; due to the limitations of a network IDS, this is the only place where the IDS receives all the traffic that comes from outside. (An alternative place would be in front of the external router; however this IDS would then detect more alerts than are actually relevant: it would include all attacks that are confounded by the exterior router.) In section 4.3 we will discuss why this topological restriction is reasonable.

If we find an attack under these restricted conditions, we will know that there is an attack in the real-world.

Figure 3.2: The network topology

3.3 Time-to-live model

In this section we present our first model. We consider whether the Internet Protocol version 4 (IPv4) [dR81] gives an attacker the opportunity to launch an undetected attack against the target. We first discuss how we can represent the protocol.

1. We need the data field, otherwise we could not communicate with the nodes in our simulated network.

2. Additionally, we include the Time-To-Live (TTL) field.

As shown in [PN98], the TTL offers an interesting evasion possibility.

3.3.1 CSP model

Each datagram consists of a TTL value and some data. Hence the channels have the following structure: $TTL.DATA$. In order to reduce the state space of our model we ought to introduce further restrictions.

- The TTL value will only range from 4 to 0. We believe that the range of the TTL value is enough, because the diameter[1] of the resulting network will be smaller then 4.

- The Data field will communicate the bit patterns A, B and C. A represents the bit patterns that, once received, force the target to move into a pre-crashed state or the IDS into a pre-alerted state. The pre-crashed or pre-alerted state indicates that the system will fail or alert on receiving a B, bit pattern (which stands for the attack suffix). If the system is not in a pre-crashed or pre-alerted state and receives a B it stays in its initial state. In the real world, A followed by B represent all possible real-world attacks that forces a target or IDS to fail or alert. A common example for an $\langle A, B \rangle$

[1]The diameter of a network, in this context, is the maximum number of time-to-live decreasing hops between the target and the attacker.

sequence would be the transmission of a buffer overflow sequence, via two packets, to a vulnerable service within the network.

The final class of bit patterns, C, represents all strings that are not part of class A or B, i.e. the set of innocent patterns that are not part of an attack in any way. We believe, that for our purpose three packets are enough to simulate every relevant pattern matching scheme. In section 4 we will show how one can formally justify that actually only 2 different patterns are required. However for this we will slightly change the focus of our model.

For simulating an appropriate network we require two routers, an attacker, one target and one IDS, as shown in Figure 3.2; we describe each of these below.

The routers The routers are used for navigating packets from source to destination. Since we have not modelled the source and destination address we can ignore the whole routing functionality; our routers, therefore, act like relay-stations. Taking this simplification into account, we achieve a very simple router that decreases the TTL field by one and checks the result. In the case where the value is zero, the packet will be dropped. Otherwise the packet will be forwarded with the new TTL value. From this, we get the following CSP description:

$$
\begin{aligned}
&Router(in, out) = \\
&\quad in?x?y \rightarrow \\
&\quad \text{if } y > 1 \text{ then } out.x.(y-1) \rightarrow Router(in, out) \text{ else } Router(in, out).
\end{aligned}
$$

where x contains the data and y the TTL value. The parameters in and out are channel names that represent the input and output ports of the router.

The attacker The attacker process should be able to execute the same actions as an attacker in the real-world. We model the attacker nondeterministically, so as to impose no limitation on the sequence of packets it sends. Consequently, FDR has to explore every possible input stream that the attacker process may create. The process is modelled by the following CSP description.

$$Attacker(out) = out.x.y \rightarrow Attacker(out).$$

The target The target process receives fragments and then reassembles them. Once the packet is reassembled, if an attack signature is found, the target should fail. The following CSP process models this component.

$$
\begin{aligned}
&Target(sigs, vulnerabilities) = \\
&\quad c?x?y \rightarrow \\
&\quad \text{let } vulnerabilities' = \{s \mid \langle x \rangle ^\frown s \in sigs \cup vulnerabilities\} \\
&\quad \text{within if } \langle x \rangle \in vulnerabilities \text{ then } fail \rightarrow Target(sigs, vulnerabilities') \\
&\qquad\qquad \text{else } Target(sigs, vulnerabilities').
\end{aligned}
$$

This process is initialised by the two variables, *sigs* and *vulnerability*. *sigs* is a set of sequences, namely all complete attack signatures. *vulnerability* keeps track of the progress of security breaches and indicates what the target has to receive in order to fail. The list comprehension is used to update *vulnerability*.

One note about the $fail$ event: this event does not mean that the computer crashes literally; it only indicates that the security policy has been violated. This can range from stealing or compromising data to root access and even to crashing.

The IDS The IDS protects the target. We assume here that the IDS is a perfect signature based IDS and therefore knows all vulnerabilities that cause the target to fail. In practice this is impossible because many vulnerabilities are not revealed yet. We have to make this assumption to generalise all current existing IDSs. The following CSP description of the IDS differs from the above target component only in the following two ways: firstly, it forwards all received packets after inspecting them; secondly, it engages in an *alert* event rather than in a $fail$ event once it has received an attack pattern.

$$
\begin{aligned}
&IDS(sigs, alerts) = \\
&\quad b?x?y \rightarrow \\
&\quad \text{let } alerts' = \{s \mid \langle x \rangle ^\frown s \in sigs \cup alerts\} \\
&\quad \text{within if } \langle x \rangle \in alerts \text{ then } alert \rightarrow c!x!y \rightarrow IDS(sigs, alerts') \\
&\qquad\qquad \text{else } c!x!y \rightarrow IDS(sigs, alerts').
\end{aligned}
$$

The complete model We use parallel composition to synchronise the different processes according to the given network structure (Figure 3.2). We hide all internal events, leaving just the *alert* and $fail$ events visible.

The Specification The specification expresses that there always has to be an *alert* before a $fail$ event. In other words, the IDS should have a log-entry once a successful attack was performed. We can model this with the following simple recursive CSP process:

$$Spec \;=\; alert \rightarrow fail \rightarrow Spec.$$

We use FDR to check whether $Spec \sqsubseteq_T Model1$ holds, that is, whether the traces of $Model1$ are a subset of the traces of *Spec*. The process *Spec* allows precisely the valid traces, so if the refinement holds then the traces of $Model1$ are just valid ones, where the IDS detects all attacks; if not, then we have discovered an attack not detected by the IDS.

3.3.2 Results

FDR reveals that the refinement check above fails and provides us with the following trace:

```
< a.A.4, b.A.3, c.A.3, a.C.2, b.C.1, d.A.2,
  c.C.1, a.B.4, b.B.3, c.B.3, d.B.2, fail >
```

This trace is displayed in the following sequence diagram presented in Figure 3.3. This is similar to the observation of Ptacek in [PN98]. The attacker sends

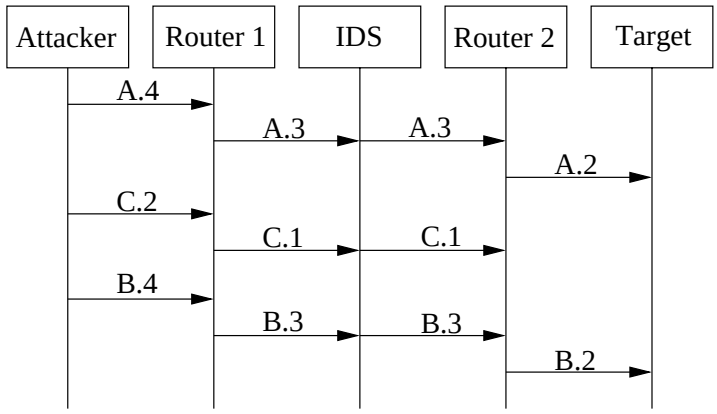

Figure 3.3: TTL Attack

three packets with data A, C and B respectively, where the packet with data C has a TTL value that is lower than its distance to the target. Therefore, this fragment will be discarded from the last router. The IDS, however, takes it into account, so the reassembled packet deviates from the packet that is processed by the target. Hence the target fails, but the IDS does not raise an alert.

Attacks like these, where the states of the IDS and target become de-synchronised, are called *de-synchronisation attacks*.

3.3.3 Discussion

The attack presented above relies on the fact that the IDS has not enough information about the topology of the network. We can solve this problem in two ways.

1. We could redesign the IDS so that it takes the different distances into account. The drawback of this solution is that the count has to be updated if changes in the network topology occur.

   We designed a CSP model corresponding to this proposed solution; the analysis found no attacks.

2. The second possibility is harder to implement; we could implement a reassembly algorithm that raises an alert if the TTL value of one fragment in the stream is different from the others, and this TTL value is lower than the

diameter of the network. On the other hand, this would cause a potential for false-positives. Let us assume the network consists of more broadcast domains (i.e. employs more routers), which means that we would have targets with different distances to the IDS. Hence it may very well be that the fragment with the low TTL value would reach the destination. To solve this, we have to link a distance to every broadcast domain. This is similar to the first solution.

Finally, we have to point out that these days it is unusual for packets to get overly fragmented; therefore an IDS based on anomaly detection is well suited for spotting an increased amount of fragments within a communication stream [SNO].

3.4 Fragment-overlapping model

In this section, we examine the behaviour of a network that includes data packets that are fragmented and reassembled according to RFC 791 [dR81].

Sometimes an IP packet has to be routed through different networks. Not all networks have the same properties. Therefore, a packet might have to be split up into fragments tagged with their position in the original packet (fragment offset); this process is termed *fragmentation*. The target receives an increased supply of smaller fragments instead of one IP packet and therefore has to reconstruct the initial packet; this process is called *reassembly*. The algorithm in [dR81] collects the fragments and puts them into the right place of the reserved buffer.

Sometimes data is received at the same fragment offset as a previously received fragment. In such a case, a decision has to be made whether to favour old or new data. RFC 791 [dR81] leaves it unspecified which should be preferred, but the recommendation is to prefer new data, so that if the algorithm receives data from the same position twice, the new data will overwrite the old.

However, not all implementations follow this suggestion: favouring new data introduces great danger, as stated in [RZT95], making it possible to circumvent filtering devices; for this reason some operating systems favour old data. Combining operating systems that favour new data (e.g. 4.4 BSD and Linux) with those that favour old data (e.g. Windows NT 4.0 and Solaris 2.6) introduces an evasion possibility if the IDS does not know what type of operating system the target is running. This was first discovered by Ptacek and Newsham [PN98].

3.4.1 CSP model

To analyse the interactions between the various types of operating systems and IDSs, we have designed the following CSP model. The network topology is similar to that in Figure 3.2, although, for simplicity, we omit the routers.

Channels The channels of this model have to be extended. We require all fields that are necessary to re-assemble the fragment stream, namely the more fragments (MF) bit, which indicates whether this is the final fragment in the packet, and the fragment offset (FO) bit, which indicates the offset of this fragment within the packet.

We use the following channel description:

$$more\_fragment\_bit.fragment\_offset.TTL.data$$

Therefore, event $a.1.0.1.A$ represents a packet that travels along channel a with its more fragment bit set to one, a fragment offset of zero, a TTL value of one, and a data field containing a bit sequence A.

Attacker The attacker process basically remains unchanged.

Target The target should satisfy the same properties as the target process of the TTL model. Additionally, it should be able to deal with fragments and out-of-order traffic. Thus, it should be able to re-assemble an out-of-order fragment stream, as it is described in RFC791.

In order to consider the behaviour of the different types of operating systems—favouring old or new data—we arrange for the target process to choose an operating system initially. The reassembly buffer is initialised to be empty ($\langle N, N, N, N, N \rangle$).

$$Target(sigs) = os\_target?os \rightarrow Target'(os, \langle N, N, N, N, N \rangle, sigs, 0).$$

The process $Target(OS, buff, sigs, max)$ requires the following parameters: OS states whether the IDS is preferring old or new data; $buff$ represents the allocated resources that are required for reassembling a packet; $sigs$ carries the set of attack signatures; and max keeps track of the maximum size of the original packet. The target first receives a datagram and calculates the new buffer $b1$ with the function $overwrite$.

It is important that our target process decides initially whether it is within the class of operating systems that favours old or new data, indicated by the events $os\_target.0$ and $os\_target.1$. After making that decision it is forced to stay in this class.

$$\begin{aligned} &Target'(OS, buff, sigs, max) = \\ &\quad in?mf?fo2?ttl?data \rightarrow \\ &\quad \text{let } b1 = overwrite(buff, fo2, data) \\ &\quad \text{within } Target''(OS, buff, sigs, max, mf, fo2, data, b1). \end{aligned}$$

The following process models the case where the more fragments flag is equal to zero, indicating that this will be the last fragment. First the process checks whether a fragment with this offset has already been received and if so, acts according to its update policy (favouring old or new data). It then checks whether or not the packet is complete. If the packet is not complete it stores the maximum size of the packet, otherwise it verifies whether it has received an attack or not. The function $nth(a, b)$ returns the value stored in position b of buffer a. $allFilled(a, b)$ checks whether all data in the buffer a up to position b have been received. $check(a, b, c)$ compares the buffer a with the set b to depth c to validate the existence of any attack patterns in the buffer. After the reconstruction of a packet the buffer is initialised with $\langle N, N, N, N, N \rangle$, which indicates a clear buffer.

$$
\begin{array}{l}
Target''(OS, buff, sigs, max, 0, fo2, data, b1) = \\
\quad \text{if } nth(buff, fo2) \neq N \wedge OS == 0 \\
\quad \text{then } Target'(OS, buff, sigs, max) \\
\quad \text{else if } allFilled(b1, fo2) \\
\qquad \text{then if } check(b1, sigs, fo2) \\
\qquad\quad \text{then } fail \rightarrow STOP \\
\qquad\quad \text{else } Target'(OS, \langle N, N, N, N, N \rangle, sigs, 0) \\
\qquad \text{else } Target'(OS, b1, sigs, fo2).
\end{array}
$$

The following process models the case where a fragment with more fragments bit set to one arrives, indicating that more fragments are following. The structure is nearly the same, except that we do not set any maximum.

$$
\begin{array}{l}
Target''(OS, buff, sigs, max, 1, fo2, data, b1) = \\
\quad \text{if } nth(buff, fo2) \neq N \\
\quad \text{then if } OS = 0 \\
\qquad \text{then } Target'(OS, buff, sigs, max) \\
\qquad \text{else } Target'(OS, b1, sigs, max) \\
\quad \text{else if } allFilled(b1, max) \wedge max \neq 0 \\
\qquad \text{then if } check(b1, sigs, max) \\
\qquad\quad \text{then } fail \rightarrow STOP \\
\qquad\quad \text{else } Target'(OS, \langle N, N, N, N, N \rangle, sigs, 0) \\
\qquad \text{else } Target'(OS, b1, sigs, max).
\end{array}
$$

The IDS The IDS process structure is similar to the IDS model in the improved TTL version in that it considers the distance to the target. It is also capable of reassembling fragments arriving out-of-order, as the target process presented above. We will not give a full account here because of the similarities to the target. The IDS raises an *alert* instead of a $fail$ event and indicates its operating system with os_ids instead of os_target.

The complete model The complete model is composed of an attacker, a target, and an IDS. The specification and refinement assertion remain the same as in the TTL example.

3.4.2 Results

FDR provided us with two distinct attacks that could both elude the IDS.

Attack 1 The IDS chooses to use an operating system that favours new data (indicated by the event $os\_ids.1$), whereas the target chooses to favour old data

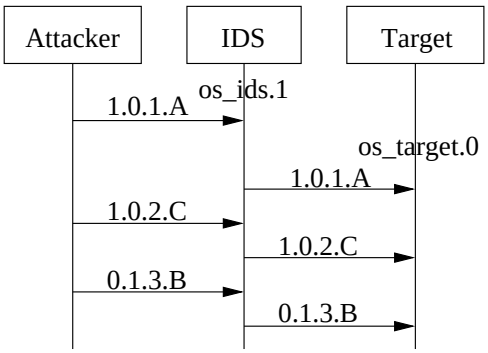

Figure 3.4: Attack 1

(os_target). The attacker sends two fragments with fragment offset zero, the first containing a bit sequence A ($1.0.1.A$), the second containing an innocent bit sequence C ($1.0.2.C$). The result is that the IDS receives the A fragment and then overwrites it with C. However, the target receives the A fragment and refuses to store the C fragment, because it favours old data. Therefore we have the situation where in the reassembly buffer of the IDS, a C bit sequence is stored and in the buffer of the target process an A bit sequence is stored. Finally the attacker creates the last packet ($0.1.3.B$), with a fragment offset of one and the more fragment bit set to zero. Hence on receiving this fragment, both the target and the IDS re-assemble their packets. The IDS re-assembles $\langle C, B \rangle$ and the target re-assembles $\langle A, B \rangle$, which causes a $fail$ event without an $alert$.

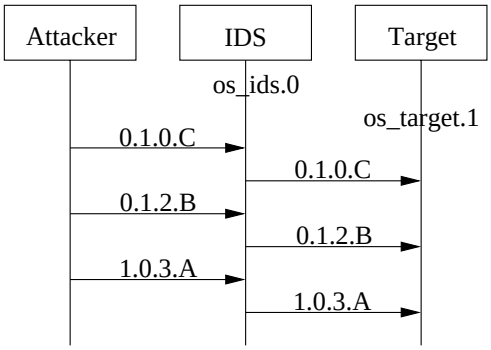

Figure 3.5: Attack 2

Attack 2 This attack is the reverse of the former. The IDS chooses to be based on a type zero OS (favouring old data) and the target chooses to be based on a type one OS (favouring new data). The attacker sends a fragment with fragment offset one, more fragment bit set to zero and a C bit sequence to the IDS ($0.1.0.C$).

He then submits a fragment with the same fragment offset but with a different bit sequence ($0.1.2.B$). This leads to a deviation of the IDS buffer from the target buffer: the IDS has stored a C in its second place, whereas the target overwrites the C with a B. The attacker then sends the final packet ($1.0.3.A$). The IDS re-assembles $\langle A, C \rangle$, which is innocent, and the target re-assembles $\langle A, B \rangle$, which leads to the $fail$ event without an $alert$.

3.4.3 Discussion

To prevent these attacks, the IDS has to take into account both possibilities for the target, i.e. favouring old or new data. We have changed the IDS accordingly. We used two parallel IDSs, one favouring old and the other favouring new data. With these changes, FDR was not able to spot any attacks. However, this solution does not appear to scale well. There are many differences in the way implementations treat the TCP/IP stacks and it would appear that the IDS needs to consider all such possibilities. Even if we only consider the drop points — operations where the packet is completely rejected — described in [PN98, Pax99], there are a vast number of de-synchronisation possibilities. The consequent state-space explosion in the IDS would appear unmanageable.

Another subtle point in our reassembly model was missing, namely time: both the IDS and target will timeout if a packet is not completely received within a certain time. The use of timeouts allows a different de-synchronisation attack. Consider the case were the timeout value of the IDS is smaller than the value for the target: the attacker then can send its first fragment and wait until the IDS times-out before sending the remaining fragments, causing the agents to become de-synchronised (this will be discussed in more detail in Chapter 5.1). This attack reveals an interesting point: we can never be certain whether our abstractions have removed details that allow attacks. Because of this we need a proper formalism to formally prove that our abstractions have not lost too much detail (see chapter 4).

3.5 Using CSP to test specifications

The models we have seen thus far reveal already known attacks [PN98]. Only some of the prevention techniques that were discussed in the previous model are new. Now we shall try to apply our knowledge in an effort to discover new ways to elude an IDS. However, we remain in the fragmentation area.

In 1982, Clark suggested, in RFC 815, a better re-assembly algorithm. This algorithm was commemorated as an improved version of the standard algorithm given in RFC 791.

We will use Clark's terminology [Cla82]. A hole is a missing bit sequence in the half-reassembled datagram. Every hole is specified by two numbers. $hole.first$ implies the starting position of the hole in the buffer. $hole.last$ describes the upper boundary of the hole. The pair of $hole.first$ and corresponding $hole.last$ is called a hole descriptor. All pairs referring to a hole are collected in the hole descriptor list. The term, $fragment.first$, is a synonym for the fragment offset. $fragment.last$ is specified by the fragment offset plus the total length minus the header length. The algorithm keeps track of its collected data with a vector of bits, called a hole descriptor table. (For further information see [Cla82].)

3.5.1 The re-assembly algorithm based on RFC 815

Every packet that arrives will be checked to see whether the arrived data affects one or more holes. Therefore, every hole in the hole descriptor list is examined. In the case that one hole is completely overlapped by the data, the corresponding descriptor will be deleted. In cases where all descriptors are deleted, the packet is re-assembled and can be processed further.

The algorithm is initiated by the arrival of the first fragment. At this point an empty buffer is allocated. The buffer should preferably be 64KB. The hole descriptor list is initialized by one descriptor going from position zero to say 64KB. The following steps are then required to insert a packet into the right position and to manage the hole descriptor list.

In the first step we choose the next hole in the descriptor list (1). In the event that there are no more entries, we proceed with step eight. Once we have a hole, we check whether *fragment.first* is greater than *hole.last* (2) or whether $fragment.last$ is less than *hole.first* (3). If one of these conditions is true, we jump back to the first step in order to choose the next hole. In the case that neither of the conditions is true, we know that the hole is affected by the arrived fragment. We therefore delete it from the hole descriptor list (4). At this point we must determine whether or not the hole is completely overlapped by the fragment. That is, we verify whether or not $fragment.first$ is greater than $hole.first$. If greater, it is deduced that we have to create a new hole starting from $hole.first$ and ending at $fragment.first$ ($hole.first$ refers to the deleted hole) (5). In step six we check whether the *hole.last* is greater than $fragments.last$. In the case

where this is true, we create a new hole from $fragment.last$ to $hole.last$ (6). Moreover, we consider the MF bit in the arrived fragment. From a zero we can infer that this is the last packet. We can, therefore, discard all hole descriptors that are pointing to successive holes. The exact quote of the RFC implies that the hole from $fragment.offset$ plus the value $fragment.last$ to the upper-bound of the buffer will be discarded. After all the calculations and updating of the hole descriptor list have been completed, we return to step one (7). The last step is reached when the algorithm times out or in step if no further holes are available. The fragment is re-assembled and can be forwarded for further processing (8). (For further information see [Cla82].)

3.5.2 Description of the RFC 791 versus RFC 815 model

As stated in [GU00], few operating system vendors have changed their algorithm. Therefore, we will most likely encounter, in a heterogeneous network, the situation where some nodes use the old method and some the new. This situation is examined in the following subsection. We will only state the result of the first CSP model of that series; the target uses the procedure according to [Cla82] and the IDS uses the original re-assembly method. A brief description of the necessary components follows.

The router has the same structure as in the former model.

The attacker has the same power as the attacker in the fragmentation overlapping mode, and therefore, is modelled in the same way.

The target uses various functions to model the re-assembly algorithm. We will describe each function in turn.

Nexthole() corresponds to the first three steps of the algorithm description. It loops through the hole descriptor list and selects the next available appropriate hole. By appropriate we mean that the checks in steps two and three are negative, which means that the hole is affected by the incoming fragment. For the special case where the list is empty, it returns the pair (-1,-1) to indicate that no hole was found. However, as we will see later, this is never the case. The resulting CSP process is:

$nexthole(holedescriptorlist, fragmentfirst, fragmentlast) =$
if $length(holedescriptorlist) = 0$
then $(-1, -1)$
else if $fragmentfirst > snd(head(holedescriptorlist)) \vee$
$fragmentlast < fst(head(holedescriptorlist))$
then $nexthole(tail(holedescriptorlist), fragmentfirst, fragmentlast)$
else $head(holedescriptorlist)$.

Delhole() is called when a fragment arrives that has the MF bit set to false. Thus, it deletes all holes that start behind the value fragment offset plus the length of that fragment. This is not stated particularly well in the RFC. It only states that we have to delete the last hole, which extends from the end of the received fragment (calculated as FO plus the length of the fragment) to the buffer boundary. However, if we implement this literally we get an altogether different vulnerability. Let us consider the following case:

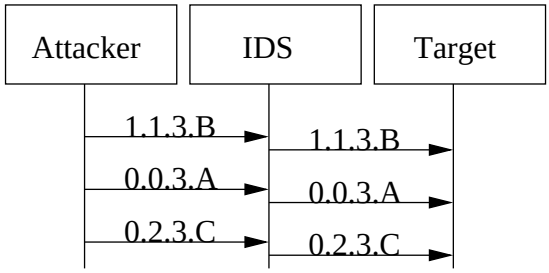

Figure 3.6: RFC 815 Example Attack

The attacker sends in Message 1 a fragment that is stored on the second place of the re-assembly buffers of the target and the IDS. Its more fragment bit is set to one, indicating that more fragments are following. Therefore the hole descriptor list of the target process contains two holes, one reaching from zero to one (hole (0,1)), and the other from 2 to 4 ((hole (2,4)), or as stated in the original description, to infinity). Message 3 then forces the IDS to re-assemble, because it has a fragment offset of zero and its more fragment bit is set to false. Furthermore, the maximum of the original IP packet is set to zero, hence the IDS only considers the first place of its buffer and only obtains A. The target receives the same packet and calls the function $Delhole()$. $Delhole()$ then tries to erase the last hole that reaches from the end of the received fragment, 1, to 4 (or infinity). However this hole does not exist. Therefore only the hole (0,1) will be deleted. At this point, the hole descriptor list contains the hole (2,3), hence the target is not able to re-assemble. Only Message 5 deletes the last hole; however,

the maximum will be changed to two, which means that the target re-assembles the bit sequence $\langle A, B, C\rangle$ and engages in a fail event.

However, for tracking down this subtle flaw we have to introduce the length. Thus far we have assumed that all bit sequences have the same length. Therefore we are unable to find attacks based on length. (This will be discussed further in the conclusion.) The CSP process for this function is defined as follows:

$$
\begin{array}{l}
delhole(holedescriptorlist, hole, n) = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle\rangle \\
\quad \text{else if } ((snd(hole)) < (fst(head(holedescriptorlist)))) \\
\qquad \text{then } delhole(tail(holedescriptorlist), hole, n-1) \\
\qquad \text{else } \langle head(holedescriptorlist)\rangle \\
\qquad\quad ^\frown delhole(tail(holedescriptorlist), hole, n-1).
\end{array}
$$

The functions, $createnewholefst()$ and $createnewholelast()$ model steps five and six of the algorithm.

The main function that models the algorithm is called $RFC815()$: It uses the functions $nexthole()$, $createnewholefst()$, and $createnewholelast()$ to delete all holes that are affected and to create the new ones in the cases where a fragment only partially overlaps a hole.

$$
\begin{array}{l}
rfc815(holedes, fragment, n) = \\
\quad \text{let } hole = nexthole(holedes, fst(fragment), snd(fragment)) \\
\quad \text{within if } n = 0 \text{ then } \langle\rangle \\
\qquad \text{else if } head(holedes) = hole \\
\qquad\quad \text{then } createnewholefst(hole, fragment) \\
\qquad\qquad ^\frown createnewholelast(hole, fragment) \\
\qquad\qquad ^\frown rfc815(tail(holedes), fragment, n-1) \\
\qquad\quad \text{else } \langle head(holedes)\rangle ^\frown rfc815(tail(holedes), fragment, n-1)).
\end{array}
$$

The last bit of CSP we must discuss is the part where we introduced the ambiguity. The RFC does not deal with the case where two fragments with the MF bit set to false appear. Therefore, the implementer has the following choices:

1. He ignores the fragment.
2. He accepts it and calculates a new packet length (new maximum).
3. He accepts it, however the length stays unchanged.
4. Because this is obviously a malformed fragment stream, he flushes the whole reassembly resources that are associated with that stream.

Therefore, at the beginning, our model chooses what it intends to do, and it behaves like this for one protocol run. The IDS still uses the RFC 791 re-assembly algorithm and thus is like option two of the target, accepting the second delimiter fragment and changing the maximum. Hence, if the target decides to choose option two, then it will behave exactly like the IDS. An attack is, therefore, impossible.

The rest of the model is straightforward and will not be explained further.

The IDS remains unchanged and, like the former model, it consists of two parallel IDSs. However, as mentioned before, the appendix also addresses the systems where the IDS operates on the new algorithm.

The channel description remains unchanged as well:

$$more\_fragment\_bit.fragment\_offset.TTL.data$$

3.5.3 Results

FDR provided us with more than one hundred possible attacks; we picked the most promising ones.

Attack No. 1

```
< os_target.0, exception.0, a.0.1.3.B, b.0.1.3.B,
a.0.0.3.B, b.0.0.3.B, a.1.0.3.A, b.1.0.3.A, fail >
```

Converting this into our message notation, we obtain the following:

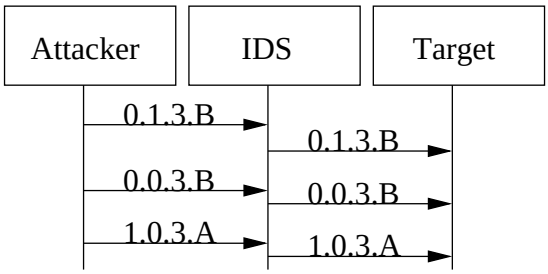

Figure 3.7: RFC 815 Attack 1

In Event 1, the target chooses to favour old data. However, as we will see this does not matter and in the subsequent Event 2 it chooses to follow the exception handling strategy zero. This is, as stated above, where the algorithm ignores the second fragment with the more fragments bit set to zero. Therefore the target

only processes Messages 3 and 7 and drops Message 5. The IDS, however, takes all three fragments into account. It receives Messages 3 and 5; at this point both IDSs are ready to re-assemble, because they receive a fragment with more fragments bit set to zero and fragment offset zero for the first time. Hence, both IDSs maintain clear buffers after processing Message 5. The target however has to wait until a fragment with fragment offset zero and more fragment bit set to one comes along. Message 7 satisfies these properties and is therefore accepted. Finally, the target re-assembles $\langle A, B \rangle$ and the IDS are maintaining a bit sequence belonging to set A in the first position of their buffers.

Attack No. 2

```
< os_target.0, exception.1, a.0.1.3.A, b.0.1.3.A, a.0.1.3.C,
b.0.1.3.C, a.1.0.3.A, b.1.0.3.A, a.0.1.3.B, b.0.1.3.B, fail >
```

More readable in message notation: With the Event 2, *exception*.1, the target

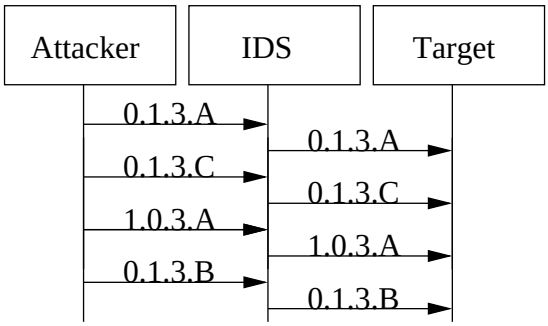

Figure 3.8: RFC 815 Attack 2

process indicates that it will — in cases where the second fragment has its more fragments bit set to zero — flush the whole buffer. The attacker uses this behaviour to elude the IDS in the following way: the attacker sends two fragments (Message 3 and 5) that cause the target to flush the re-assembly buffer. The IDS, favouring old data, stores an A on the second position of its buffer while the other IDS stores a C on the same position. Message 7 causes the IDS to re-assemble, therefore one IDS obtains $\langle A, A \rangle$ and the other obtains $\langle A, C \rangle$. The buffer of the target maintains an A on its first position, and on receiving Message 9, the target re-assembles the bit sequence $\langle A, B \rangle$.

Attack No. 3

```
< os_target.0, exception.3, a.0.1.3.B, b.0.1.3.B, a.0.0.3.A,
b.0.0.3.A, fail >
```

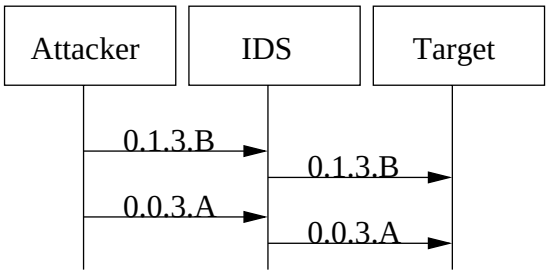

Figure 3.9: RFC 815 Attack 3

We convert this trace again into the corresponding easy-to-read message notation, presented in Figure 3.9.

At this point the target decides to accept a second fragment with its more fragment bit set to zero. However, the amount of data (the maximum) that needs to be received stays unchanged. In Message 3 the attacker designs a fragment that indicates that the overall packet was of size two with the more fragments bit set to zero and its fragment offset set to one. Hence, the maximum of the IDS and the target is one. This means that they have to receive only one more packet with fragment offset zero. On receiving Message 5 however, the IDS changes its maximum to zero, whereas the maximum of the target remains unchanged. Hence, the IDS only considers the first position of their buffer and re-assembles the sequence A. The target still has a maximum of one and therefore takes positions one and two of its buffer into account — it re-assembles the sequence $\langle A, B \rangle$.

3.5.4 Discussion

Inaccurately written RFCs can be interpreted in different ways and the resulting ambiguities can have serious effects upon the security of a system. The best method to define such routines is, as done in RFC 791, to enforce an example implementation. Even in this very strict RFC, the only choice a programmer had, choosing whether his implementation would favour old or new data, caused a vulnerability. As we have shown, this small interpretation gap was sufficient to introduce a security hole. Furthermore, we have seen the impact that the addition of one field (in this particular case the length) of the IP header can introduce further possibilities to de-synchronize the IDS. However, a model that simulates the length of fragments is not provided.

3.6 Conclusion

We have seen how small deviations in implementations can have a considerable impact on the security of a system. Even when the individual subcomponents of a system are secure, the overall system may be still not free from flaws. Such emergent faults can be spotted easily by testing the system as a whole against a specification.

Sections 3.3 – 3.5 established that CSP is suitable to verify IDSs. We considered the reproduction of known attacks and then applied our knowledge to other scenarios within the intrusion detection area.

In section 3.3, we produced a CSP model that showed how FDR can be used to detect emergent faults. The core of the model represented a simplified version of IPv4, only consisting of a payload and a time-to-live field. We showed that this simple protocol model hides known attacks. We concluded this section by discussing various solutions.

In section 3.4 the TTL model was enhanced to cover the reassembly functionality of IPv4. Reassembly is required whenever a packet, larger then the maximum transfer unit[2] of the destination network, travels from one network into another. Hence the protocol consists of offset, time-to-live, payload and the more-fragment-bit field. The reassembly procedure is based on RFC 791. Finally we addressed the solvability of the detected problems and suggested improvements.

In section 3.5 we extended this protocol structure by adding another reassembly technique to our network. The system comprised hosts that supported reassembly algorithms according to RFC 791 as well as RFC 815. Since RFC 815 does not provide an example implementation, we generated a process that included all reasonably possible interpretations of the RFC.

Generalising the retrieved results Whenever it is possible to create a difference between the input stream of the IDS and the protected system we can successfully hide an attack. More generally, both the protected system and the IDS have state transition graphs; if we create a situation where these systems change into different states, they require different stimuli to reach the deprecated state where the target fails and the IDS raises the alert. We can distinguish between three de-synchronising possibilities:

1. De-synchronisation due to the systems behaving exactly the same, but the input streams being different; this is the method that appears in our first model.

2. De-synchronisation as a result of the input streams being the same, while

[2]The maximum transfer unit equals the maximum length of a packet that is allowed to travel through the network without being fragmented.

the systems behave differently under certain conditions; this is the type of flaw exploited in our second and third model.

3. De-synchronisation because both the input streams and the behaviour of the systems are different.

Chapter 4

Towards a more complete analysis

The disadvantage of using tools like FDR is that they search every state of the process and hence the state graph of the process under investigation has to not only be finite but also reasonably small. In order to keep our model below a certain complexity we have to abstract away details. We did so by applying abstractions on the general composition of the model, for example modelling fewer fields of IPv4, and by limiting the scope of the data types involved. However by doing so, it remains uncertain whether these restrictions abstract away other specification violations. Hence, the investigation in Chapter 3 is not finished. We do not know whether the inability to spot attacks in the improved models stems from over-abstraction or because there really is no attack.

In this Chapter we generalise our results so that we are able to verify the model for *all* values of the type of network packets and the set of attack signatures. We change the focus of the time-to-live model from section 3.3, to obtain a more complete analysis, independent of the set *sigs* of attack signatures. Finally, we generalise the parameters of the system further. We show that the range $\{1\,..\,3\}$ for the TTL field is sufficient to capture all essentially-distinct behaviours of the system for arbitrary TTL values. Additionally we prove that a buffer size of five (in the buffer-reassembly model) is sufficient and that the network topology, which at first sight seems very restricted is sufficiently general.

Finally we will describe an algorithm that automatically shows whether the generated counterexample also exists in the real world system.

4.1 Generalising the types of packets and signatures

We now change the focus of the time-to-live model from Section 3.3, in order to move towards a more complete analysis, independent of the set *sigs* of at-

tack signatures. We observe that the IDS is really doing two things: filtering out packets that will not reach the target and performing pattern matching on the remaining packets. It is therefore possible to split the IDS into two different processes corresponding to these functions. In related models, such as the packet reassembly model in Section 3.4, the target process similarly performs a combination of filtering and pattern matching, and so it is possible to split the target into two processes. This gives the topology presented in Figure 4.1.

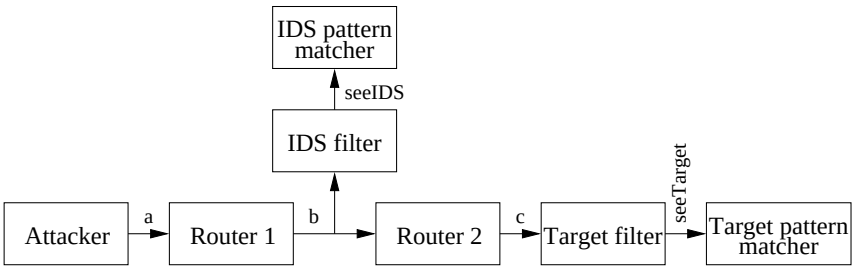

Figure 4.1: Network topology for the revised time-to-live model

We can then make the following observation:

Observation 1 *If the stream of packets passed to the pattern matching component of the IDS is the same as the stream of packets passed to the pattern matching component of the target, then the IDS will detect all attacks.*

The hypothesis of Observation 1—that the two components see the same stream of packets—is an easy property to test. In fact it does not even require us to model the two pattern matching components; simply the messages passed to them on the channels $seeIDS$ and $seeTarget$ suffice. The advantage of this change of focus is that is allows us to remove the parameter $sigs$ from the model, thus leading towards a more general verification.

The two filtering processes (where the IDS takes the distance to the target into account) can be modelled by:

$$\begin{aligned} IDS &= b?x?y \rightarrow \text{if } dist \leq y \text{ then } seeIDS.x \rightarrow IDS \text{ else } IDS, \\ Target &= c?x?y \rightarrow seeTarget.x \rightarrow Target. \end{aligned}$$

where $dist$ is the distance from the IDS to the target. The rest of the network is unchanged.

It is easy to capture the hypothesis of Observation 1 as a refinement assertion; there is a certain amount of buffering in the system, and the specification has to take this into account:

$$\begin{aligned} Spec = & \prod_{x:T} (seeIDS.x \rightarrow Spec'(x) \sqcap seeTarget.x \rightarrow seeIDS.x \rightarrow Spec) \\ & \sqcap \\ & STOP, \end{aligned}$$

$$Spec'(x) = seeTarget.x \rightarrow Spec \sqcap \sqcap_{y:T} seeIDS.y \rightarrow seeTarget.x \rightarrow Spec'(y).$$

The specification captures the property that the IDS and target see the same stream of packets, except that the IDS might at any point have seen up to two more packets than the target, or the target might have seen one more packet that the IDS.

We can then use FDR to check that the system failures-refines *Spec*. The refinement holds, and we can then use Observation 1 to deduce that the IDS detects all attacks *for all possible sets of attack signatures*.

However, this appears to leave us only slightly better off than before: we can verify the system for a fixed type *DATA*; but does this tell us anything about systems with different values for *DATA*? It turns out that we can use Theorem 1 to show that this is indeed the case. The system and specification processes are both data independent with respect to the type *DATA* (when the pattern matching against attack signatures was included, the IDS and target were not data independent); both satisfy NoEq_{DATA}; and the specification satisfies Norm_{DATA}. The Theorem therefore tells us that we only have to check the refinement for a type *DATA* of size 2, say $DATA = \{A, B\}$, to have verified it for all values of *DATA*.

4.2 The packet reassembly model

The same technique can be used for our second example, presented in Section 3.4. We can model the reassembly algorithms, at both the target and IDS, as separate processes. We can then ask whether the stream of packets passed to the pattern-matching components are the same, using essentially the same specification as for the time-to-live model. This generalises our analysis to all possible sets of attack signatures. We can then use Theorem 1 to show that we need only perform this analysis for a type *DATA* of size 2 to deduce that the same result holds for all larger types.

4.3 Remaining points

In this section we generalise the parameters of the system further. We show that the range $\{1 \,..\, 3\}$ for the TTL field is sufficient to capture all essentially-distinct behaviours of the system for arbitrary TTL values. We further show that a buffer size of five (in the buffer-reassembly model) is sufficient. We also discuss why the network topology is not as restricted as it appears to be.

4.3.1 The TTL-value range

Consider the network in Figure 3.2. It is obvious that the diameter of the network, counted in TTL-decreasing hops, is two. Hence there appears to be no need to consider TTL values greater than three. We formalise this argument in this section.

Let $System(N)$ be the CSP process representing a system using TTL values $\{1..N\}$, and let $System_0(N)$ be the corresponding process before internal events are hidden, i.e. such that all channels are visible.

The technique we use is to define a collapsing function ϕ over events, that maps behaviour of a system using arbitrary-sized TTL values to corresponding behaviours of the small system[1]:

$$\forall\, tr : \Sigma^* ; X : \mathbf{P}\,\Sigma \bullet (tr, \phi^{-1}(X)) \in failures(System_0(N)) \Rightarrow (\phi(tr), X) \in failures(System_0(3)). \tag{4.1}$$

We define ϕ over events, to replace TTL values greater than 3 on the channel a by the value 3, and similarly for other channels:

$$\begin{aligned} \phi(a.x.y) &= \text{if } y \leq 3 \text{ then } a.x.y \text{ else } a.x.3, \\ \phi(b.x.y) &= \text{if } y \leq 2 \text{ then } b.x.y \text{ else } b.x.2, \\ \phi(c.x.y) &= \text{if } y \leq 1 \text{ then } c.x.y \text{ else } c.x.1, \end{aligned}$$

and ϕ is the identity function over all other events. We lift ϕ to traces, point-wise.

It is then straightforward to see that each component of the system respects ϕ — i.e. such that, analogously to equation (4.1), if $(tr, \phi^{-1}(X))$ is a behaviour of the component with N TTL values, then $(\phi(tr), X)$ is a behaviour of the component with 3 TTL values. Hence equation (4.1) itself is satisfied. However, within $System$, the channels a, b and c are hidden, so ϕ is the identity function over the visible events, and we deduce

$$failures(System(N)) = failures(System(3)).$$

But we already know that $Spec \sqsubseteq_F System(3)$; hence we can deduce that $Spec \sqsubseteq_F System(N)$, for all N.

4.3.2 Buffer size

We now argue that in the reassembly model it is sufficient to consider a buffer size of five, in the sense that if there is an attack upon a system that uses a larger buffer, then there is also an attack upon a system that uses a buffer of size five.

Suppose there is an attack trace tr in the general case. We map this onto an attack trace tr' in the restricted case of buffer size five; we do this by uniformly

[1] Σ represents the set of all events; $\phi(tr)$ represents ϕ applied point-wise to trace tr; $\phi^{-1}(X)$ represents the inverse image of X under ϕ.

remapping fragment offsets onto the set $\{0\,..\,4\}$. The construction is slightly complicated by the fact that we need to ensure that the target and IDS reassemble packets at the same points in tr' as they did in tr.

Suppose, firstly, that the attack trace tr is such that the IDS and target reassemble the initial packet differently. We arrange that in tr' they again reassemble the initial packet differently. We perform a case analysis.

Case 1 The IDS and target reassemble packets of different sizes, m and n respectively. Without loss of generality suppose $m < n$. Suppose the last fragments of those packets to be received by the IDS and target (i.e., last according to the order of the trace, as opposed to necessarily the fragments with the greatest offsets) are at offsets x and y respectively. We perform a further case analysis:

- Case x, y, m, n all distinct, with $x, y < m < n$. We define the remapping function ϕ over fragment offsets as follows:

$$\begin{aligned}\phi(i) \;=\; &\text{if } i = x \text{ then } 1 \text{ else if } i = y \text{ then } 2\\ &\text{else if } i = m \text{ then } 3 \text{ else if } i = n \text{ then } 4 \text{ else } 0.\end{aligned}$$

  We then lift ϕ to events by, for example

$$\phi(a.mf.fo.ttl.data) \;=\; a.mf.\phi(fo).ttl.data.$$

  Consider the effect of applying ϕ point-wise to all communications on channels other than $seeIDS$ and $seeTarget$. It is then easy to see that the routers and the attacker respect ϕ. It is also easy to see that the IDS and target reassemble different packets: the IDS reassembles a packet of size 4 when it receives the fragment with offset 1; and the target reassembles a packet of size 5 when it receives the fragment with offset 2.

- Case otherwise. One can construct a similar remapping function in each case, mapping onto a proper subset of $\{0\,..\,4\}$ if some of x, y, m, n coincide.

Case 2 The IDS and target reassemble packets of the same size, n say, but that differ at some point, say at offset z. Again suppose the last fragments of those packets to be received by the IDS and target are at offsets x and y respectively.

- Case x, y, z, n all distinct. We define the remapping function ϕ over fragment offsets as follows:

$$\begin{aligned}\phi(i) \;=\; &\text{if } i = x \text{ then } 1 \text{ else if } i = y \text{ then } 2\\ &\text{else if } i = z \text{ then } 3 \text{ else if } i = n \text{ then } 4 \text{ else } 0.\end{aligned}$$

  We lift ϕ to events as above. Then it is easy to see that the IDS and target both reassemble packets of size 4, after receiving the fragments with

offsets 1 and 2 respectively. However, these packets differ at offset 3: the two components receive the same sequence of data packets at this offset as they received at offset z in the general case, and so they will, as in the general case, reassemble packets that differ at this offset.

- Case otherwise. All other cases are similar.

Suppose now that the IDS and target reassemble a packet other than the first differently, say the nth packet. Then the above construction will either cause the IDS and target to reassemble a packet earlier than the nth differently (introducing a new undetected attack); or the construction will leave the first $n-1$ packets matching, but again cause them to construct different nth packets.

Finally the case where one agent reassembles a packet and the other does not can be treated in a similar manner.

4.3.3 Network topology

We have considered only a single, fixed, network topology. This turns out to be an important consideration. If there are two or more routes between the IDS and the target, then it is possible for the IDS and target to see fragments in a different order, which opens up the possibility of another desynchronisation attack. However, most local networks have only a single route from the gateway to each host, which prevents such attacks. Basically every corporate network that has more routers, which would allow multiple routes, is protected by a Firewall that enforces packets to be transmitted only in one certain way (i.e. the strict IP source routing option is prohibited). Under this condition, we believe that our abstraction has not lost any attacks:

- It is safe to abstract away from the parts of the network before the IDS, because the attacker can effectively choose what fragments are sent past the IDS (channel b in the earlier models).

- It is also safe to abstract away those parts of the local network not on the direct route from the IDS to the target, because they do not affect what the target sees.

- The remaining issue is the number of routers between the IDS and the target; there does not seem to be any intrinsic difference between one router reducing the TTL field by one, and N routers reducing the TTL by N, because the attacker is able to choose the TTL appropriately; however, the number of routers does affect the buffering of the system and hence the appropriate specification process.

4.3.4 Protocol abstractions

Finally, we have, of course, abstracted away from many of the details of the underlying network protocol. Our long-term aim is to remove these abstractions, so as to model the whole of the Internet Protocol version 4 (6).

4.3.5 Verification of abstract counterexamples

Usually it is trivial to see whether a discovered attack is a remnant of our abstraction (a false -positive) or a true attack — that also exists in the real world. However, considering that FDR can reveal 100 counter example at once it would be preferable to have a formalism that distinguishes between true and false attacks.

Let us assume that M is an (infinite-state) description of our system, and A is the result of our abstraction (finite-state). Further let's name the function that maps behaviours of M to those of A ϕ.

This mapping function has to satisfy following properties:

- for every behaviour tr of M, $\phi(tr)$ is a behaviour of system A
- if tr is an attack in M, then $\phi(tr)$ is an attack in A.

However, for our purpose we have to facilitate ϕ the other way around. Given an attack tr' on A, we have to determine whether there is a behaviour tr of M, such that $\phi(tr) = tr'$, and such that tr is an attack. For this, we only need to consider sufficient behaviours tr such that $\phi(tr)$ equals tr' and test whether they are indeed attacks on M. This task could be achieved by employing a simulator for M.

This solution is only possible if we can find a finite set of traces that are indeed sufficient to come to a conclusion. Given that following assumptions about our, on data independence based, abstraction hold — a finite set of traces is enough:

Let's first consider the common case where ϕ is defined to collapse some data independent type T. i.e., we

- start off with a function $\phi_T : T \to T$;
- lift ϕ_T to a function $\phi_\Sigma : \Sigma \to \Sigma$ by $\phi_\Sigma(c.x) = c.\phi_T(x)$ (for x of type T);
- lift ϕ_Σ pointwise to traces, i.e. $\phi(\langle a_1, ..., a_n \rangle) = \langle \phi_\Sigma(a_1), ...\phi_\Sigma(a_n) \rangle$.

If the system is data independent with respect to some type T, then it should satisfy the following property:

> If M accepts the event $c.x$, where x is of type T, and x is fresh (i.e. hasn't appeared earlier in the trace, and is distinct from all constants in the initial system and specification), and the behaviour after this

> event is described by the process P, then M should also accept $c.y$, for all fresh y of type T, and the subsequent behaviour is described by $P[y/x]$.

In other words M treats all fresh values of type T equivalently. Thus we can try to find tr as follows.

Suppose we've already found some trace tr_0 of M that corresponds to some prefix tr'_0 of tr' (i.e. $\phi(tr_0) = tr'_0$), and let's suppose the next event of tr' is $c.z$. Now we need to consider events $c.x$ of M such that $\phi_T(x) = z$. The previously stated quote shows that M acts in the same way after all such $c.x$ for fresh values x; hence we only need to consider one value. However, we also need to consider all non-fresh values w such that $\phi_T(w) = z$ (i.e. all w that have either already occurred, or that were constants in the original system or specification).

This leads to the following algorithm:

> *Given a state M' of M, and a suffix $\langle c.z \rangle^\frown tr''$ of tr',*
> *find a trace of M' corresponding to $\langle c.z \rangle^\frown tr''$ as follows :*
> *FOR every non − fresh w such that $\phi_T(w) = z$,*
> *and for a single fresh x such that $\phi_T(x) = z$*
> *DO*
> *IF M' can communicate that value on c* then
> *let M'' be the subsequent state of M'*
> *recursively try to find a trace of M'' corresponding to tr'*
> *END.*

This algorithm is finitely branching, and the trace is finite, so we have to consider finitely many traces of M. In fact, if the number of values of type T in tr' is n, and the number of constants is c, then we have to consider at most $(n + c)!/c!$ traces of M.

4.4 Summary

We showed techniques that could close the gap between our abstracted model and the real world. The necessity to reduce the complexity of the real world lies in the working principle of FDR; it verifies every possible state to the process. The abstractions themselves were conducted on the general structure of the model, by using a specific network topology, and by restricting the scope of the modelled fields of the IPv4. These techniques prevented us from drawing conclusions about whether the real world scenario behaves the same; thus it remained uncertain whether our inability to find more attacks was a result of our simplifications or whether there were indeed not more vulnerabilities.

We changed the focus of the time-to-live model, to accomplish a more complete analysis. We showed that it is enough for the data type of the payload to be a set containing only two distinct members.

Additionally we showed that the diameter of the network as the range for the TTL data type is suitable to draw conclusions upon the completeness of the FDR analysis. Furthermore we showed that it is valid to restrict the reassembly buffer size to five. The Chapter was concluded by a discussion about the network topology and other protocol abstractions.

We are aware that between our initial models and the models we used to show the absence of vulnerabilities exists a little gap. Since the proofs provided work only on the modified models we can not be certain whether the abstractions applied to our initial models did not cover attacks. However, looking at the two types of models the gap seems to be sufficiently small to be neglected. The first type of models (section 3.3, 3.4 and 3.5) is intended to find attacks and to provide easy-to-reuse counterexamples. The second type of models, provided in this chapter, is intended to show that our work-arounds are sound and that there are indeed no attack possibilities left.

Finally, we discussed an algorithm that automatically determined whether a discovered attack was a false positive or a true attack.

Chapter 5

Unexpected timing issues

Today's IDS testing is performed by a *trial-and-error* approach. This approach lacks proper coverage of timing issues. However on further inspection of the protocols that are used in today's networks, we find timeout methods everywhere. Not taking time into account has proven to be a serious mistake.

We will use two approaches for representing time: first we will design an easy-to-build CSP model using the sliding choice operator; and second we describe a discrete time CSP model by using the event *tock* to represent the passage of a certain amount of time. The models are based on the fragment-overlapping model and are kept as simple as possible.

We conclude with a discussion about the relationship between the discrete untimed and the untimed model. We will show that whenever the easy-to-use untimed model contains no vulnerabilities then the more complicated discrete model does not either. From this specific result a more general way of reasoning about the relationship between discrete and untimed timeout models is derived. This chapter also contains the formal foundation that shows that whenever the untimed model refines the specification then the tocks-based discrete time model does also. Thus giving us the opportunity to verify the process using only the simpler version of timeouts without fearing missing a specification violation in the more complex discrete time model. This holds for all untimed safety specifications.

5.1 Using the interrupt operator for simulating timing issues

The interrupt operator or the sliding choice is the easiest way to model the passage of time in order to spot vulnerabilities based on timing issues. In this section, we describe the components and the results we obtained from that model. We also discuss the impact of these results on the security of an IDS infrastructure.

5.1.1 Components

This model is based upon the fragment overlapping model. The router process remains unchanged. The possibility that the router can explicitly delay packets is not included. However, we will see that this is not required.

IDS and target are using the sliding choice operator. The sliding choice operator offers the choice between, either accepting a new input after receiving a datagram or flushing the already stored information.

$$
\begin{aligned}
&Target(OS, buf buff, sigs, max) = \\
&\quad (in?mf?fo2?ttl?data \rightarrow \\
&\quad \text{let } b1 = overwrite(buff, fo2, data) \\
&\quad \text{within } Target'(OS, buff, sigs, max, mf, fo2, data, b1)).
\end{aligned}
$$

The first part of the target process ($Target$) remains unchanged, since we do not allow the reassembly mechanism to timeout before or while processing a datagram. The real difference to our previous models becomes apparent in the processes $Target'$. After processing the current datagram, instead of evoking the process $Target$ to accept another input, we call the process $ResetTarget$.

$$
\begin{aligned}
&Target'(OS, buff, sigs, max, 0, fo2, data, b1) = \\
&\quad \text{if } nth(buff, fo2)! = N \wedge OS = 0 \\
&\quad \text{then } (ResetTarget(OS, buff, sigs, max) \\
&\quad \text{else if } allFilled(b1, fo2) \\
&\qquad \text{then if } check(b1, sigs, fo2) \\
&\qquad\quad \text{then } fail \rightarrow STOP \\
&\qquad\quad \text{else } Target(OS, \langle N, N, N, N, N \rangle, sigs, 0) \\
&\qquad \text{else } ResetTarget(OS, b1, sigs, fo2).
\end{aligned}
$$

$$
\begin{aligned}
&Target'(OS, buff, sigs, max, 1, fo2, data, b1) = \\
&\quad \text{if } nth(buff, fo2) \neq N \\
&\quad \text{then if } OS = 0 \\
&\qquad \text{then } ResetTarget(OS, buff, sigs, max) \\
&\qquad \text{else } ResetTarget(OS, b1, sigs, max) \\
&\quad \text{else if } allFilled(b1, max) \wedge max \neq 0 \\
&\qquad \text{then if } check(b1, sigs, max) \\
&\qquad\quad \text{then } fail \rightarrow STOP \\
&\qquad\quad \text{else } Target(OS, \langle N, N, N, N, N \rangle, sigs, 0) \\
&\qquad \text{else } ResetTarget(OS, b1, sigs, max).
\end{aligned}
$$

For a more detailed description of this processes see Section 3.4. There is no need to elaborate on the CSP code of the IDS process. The structure is absolutely identical, apart from different names for the processes. The $ResetTarget$ process uses the sliding choice operator to decide whether or not the next packet arrives

before the timeout triggers. It re-initialises the reassembly processes with their starting values or it accepts a new packet.

$$\begin{aligned} &ResetTarget(OS, buff, sigs, max) = \\ &\quad Target(OS, buff, sigs, max) \\ &\quad \triangleright flushTarget \rightarrow Target(OS, \langle N, N, N, N, N \rangle, sigs, 0). \end{aligned}$$

The sliding choice operator $(P \triangleright Q)$ itself is semantically equivalent to $(P \sqcap STOP) \,\Box\, Q$: it gives the environment a choice if the environment is quick enough to choose it. The Specification remains unchanged since we are hiding the events $flushIDS$ and $flushTarget$.

$$Spec = alert \rightarrow fail \rightarrow Spec$$

The overall system differs from the original TTL model only in the additional events that have to be hidden, such as $flushIDS$ and $flushTarget$.

5.1.2 Result

After applying FDR to evaluate the refinement we obtain the following trace.

```
< os_ids.1, a.1.0.1.A, b.1.0.1.A, os_target.1, c.1.0.1.A,
_tau, d.1.0.1.A, _tau, a.0.1.1.B, flushIDS, b.0.1.1.B,
c.0.1.1.B, d.0.1.1.B, fail >
```

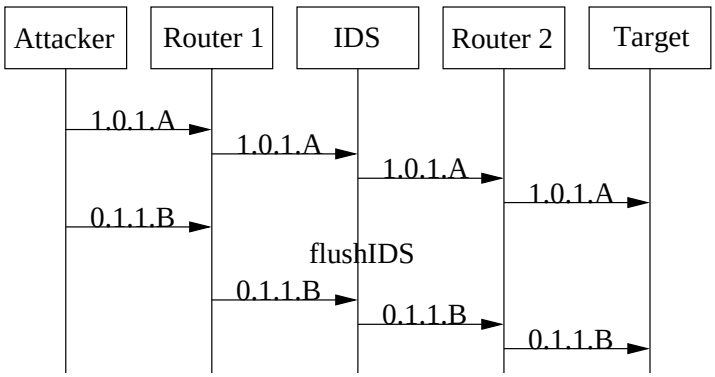

Figure 5.1: Discrete-Time Attack 1

The attacker sends the first part of his attack (bit sequence A). The datagram travels through the DMZ, finally reaching the internal router (indicated by event $c.A$). As soon as the IDS has processed the attack prefix, it moves into the pre-alerted state. It then waits for another input. However, the attacker delays the next input and waits until the IDS times-out, thus flushing its buffer. The

attacker then proceeds with sending. The remainder of his attack (the sequence B). Meanwhile, the internal router forwards the attack prefix to the target. The target moves into a pre-crashed state. Since the IDS has lost its information about receiving an A, it will interpret the fragment B as innocent and will therefore stay in its initial state. The target, however, eventually fails. Clearly, there is also the counterpart of this attack, where the target times out and the IDS does not. After a timeout, the IDS needs a different input to raise an alarm, followed by the target failing, since the IDS starts in a different state than the target.

5.1.3 Discussion

The spotted attack can be interpreted in two ways. The most obvious one is that the timeout values of the IP reassembly algorithms are equal, and the internal router delays the forwarding process of the first packet until the IDS times-out. The other way of interpreting this trace can be that the timeout value of the IDS is too short to receive both fragments. This demonstrates one danger of this model. The same trace can have multiple reasons, thus it can easily be that we miss a security hole. On the other hand this type of model is easy to create — the only problem that we are faced with is where to put the sliding choice operator. However, in our case, the RFC 791 leaves no room for interpretation. It has to appear after receiving and processing a datagram. To overcome the ambiguity of the retrieved traces, we use the discrete-time CSP model.

5.2 Discrete-time model

This model uses the event *tock* to symbolise the passage of time. After engaging in a non-*tock* event, the process offers the *tock* event and the next non-*tock* event. Thus it gives the system the option to let time pass between its activities. The resulting descriptions of the CSP processes are slightly more complicated then in the previous model.

5.2.1 Components

The attacker process can either send a packet or can allow time to pass. In this model it may be the case that an attacker could execute infinitely many actions without time passing. However, since we do not specify how long a *tock* is, this seems acceptable.

$$\begin{aligned} &Attacker(out) = \\ &\quad out.w.x.y.z \rightarrow Attacker(out) \\ &\quad \Box\, tock \rightarrow Attacker(out). \end{aligned}$$

We have decided to model two different kinds of routers, one with and the other without delay. The router without delay, process *Router*, is similar to the attacker: it has the choice to forward a packet or to engage in a *tock* event.

$$\begin{aligned}&Router(in, out) = \\ &\quad in?w?x?y?z \rightarrow out!w!x!y!z \rightarrow Router(in, out) \\ &\quad \Box\ tock \rightarrow Router(in, out).\end{aligned}$$

The router with delay, process $Router2$, can act similarly; with the exception that once it receives a datagram, it can forward instantaneously or can delay it to allow time to pass.

$$\begin{aligned}&Router2(in, out) = \\ &\quad in?x \rightarrow Router2'(x, in, out) \\ &\quad \Box\ tock \rightarrow Router2(in, out)\end{aligned}$$

$$\begin{aligned}&Router2'(x, in, out) = \\ &\quad out!x \rightarrow Router2(in, out) \\ &\quad \Box\ tock \rightarrow Router2'(x, in, out).\end{aligned}$$

The IDS and target processes are built upon the same structure as the original fragment overlapping model; hence we will only describe the extension that implements the time-out. Note, if we consider no parallel packet processing, there is, according to RFC 791, no reset during the processing of a fragment. Thus, we have omitted a *tock* event while the reassembly buffer is updated. The target is initialised by the variable *timeout*, which indicates the number of tocks that have to pass until the process flushes the reassembly buffer.

$$\begin{aligned}&Target'(OS, buff, sigs, max, 0, fo2, data, b1, timeout) = \\ &\quad \text{if } nth(buff, fo2) \neq N \wedge OS = 0 \\ &\quad \text{then } ResetTarget(OS, buff, sigs, max, timeout, timeout) \\ &\quad \text{else if } allFilled(b1, fo2) \\ &\qquad \text{then if } check(b1, sigs, fo2) \\ &\qquad\quad \text{then } fail \rightarrow STOP \\ &\qquad\quad \text{else } Target(OS, \langle N, N, N, N, N\rangle, sigs, 0, timeout) \\ &\qquad \text{else } ResetTarget(OS, b1, sigs, fo2, timeout, timeout).\end{aligned}$$

$$
\begin{aligned}
&Target'(OS, buff, sigs, max, 1, fo2, data, b1, timeout) = \\
&\quad \text{if } nth(buff, fo2) \neq N \\
&\quad \text{then if } OS = 0 \\
&\qquad \text{then } ResetTarget(OS, buff, sigs, max, timeout, timeout)) \\
&\qquad \text{else } ResetTarget(OS, b1, sigs, max, timeout, timeout) \\
&\quad \text{else if } allFilled(b1, max) \wedge max \neq 0 \\
&\qquad \text{then if } check(b1, sigs, max) \\
&\qquad\quad \text{then } fail \rightarrow STOP \\
&\qquad\quad \text{else } Target(OS, \langle N, N, N, N, N \rangle, sigs, 0, timeout) \\
&\qquad \text{else } ResetTarget(OS, b1, sigs, max, timeout, timeout).
\end{aligned}
$$

The process *ResetTarget* implements the timeout module. The variable *tocks* holds the number of tocks that are left until the target returns into its initial state and *timeout* harbors the initial timeout value. *ResetTarget* can engage in *tock* events and eventually flush the buffer of the target, or it can receive new packets without timing out. In the latter case, it resets the variable *tocks*, since a new timeout period has to start after processing the received datagram.

$$
\begin{aligned}
&ResetTarget(OS, buff, sigs, max, timeout, tocks) = \\
&\quad tock \rightarrow \text{if } tocks = 1 \\
&\qquad \text{then } flushTarget \rightarrow Target(OS, \langle N, N, N, N, N \rangle, sigs, 0, timeout) \\
&\qquad \text{else } ResetTarget(OS, buff, sigs, max, timeout, tocks - 1) \\
&\quad \Box\ Target(OS, buff, sigs, max, timeout).
\end{aligned}
$$

Since the target process and the IDS are nearly the same, there is no need to elaborate on the CSP code of the IDS. The IDS works with the same time-out mechanism. The specification as well as the overall system remain unchanged.

5.2.2 Results

We initialised the IDS and the target with timeout values of 1 and respectively 2. Further, we used the router without the delay functionality. FDR provided us with following counter example:

```
< os_ids.1, a.1.0.1.A, b.1.0.1.A, c.1.0.1.A, os_target.1,
d.1.0.1.A, tock, a.0.1.1.B, flushIDS, b.0.1.1.B, c.0.1.1.B,
d.0.1.1.B, fail >
```

This reassembles the second interpretation of the attack found in the previous model. The IDS processes the attack prefix and flushes the buffer afterwards, whereas the target does not flush its buffer. This allows the target to receive and process the complete attack sequence $\langle A, B \rangle$.

The other way of interpreting the trace of our last model was to assume that both timeout values are equal, but the components between the target and the IDS were delaying some fragments. To determine whether this line of reasoning

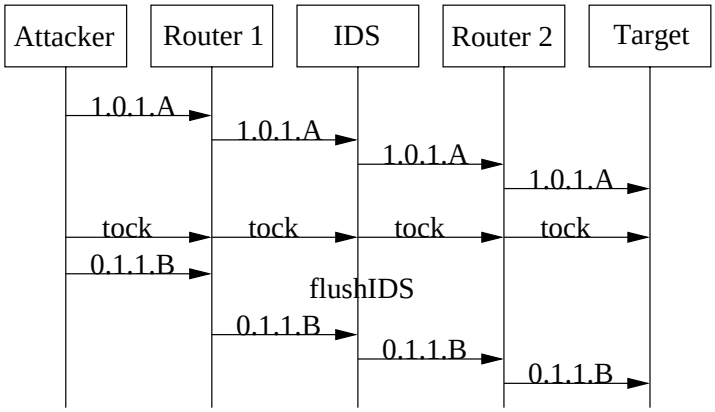

Figure 5.2: Discrete-Time Attack 1

holds, we initialise the *timeout* variables of both, the IDS and the target with the same value and use the router with the delay functionality. The refinement fails and FDR finds the following counterexample.

```
< os_ids.1, os_target.1, a.1.0.1.A, b.1.0.1.A, c.1.0.1.A,
tock, d.1.0.1.A, tock, flushIDS, a.1.1.1.B, b.1.1.1.B,
c.1.1.1.B, d.1.1.1.B, fail >
```

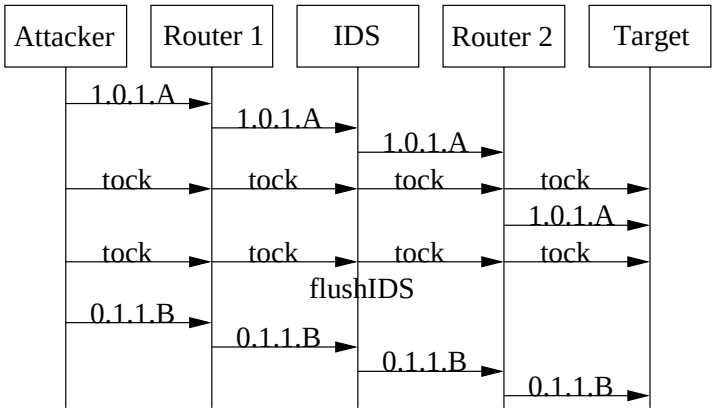

Figure 5.3: Discrete-Time Attack 2

The attacker starts by sending the first half of his attack. The fragment $\langle A \rangle$ travels through the network until it reaches the internal router. The router delays the fragment until the IDS flushes its buffer. Thus, the IDS looses its information about already received fragments. Meanwhile, the attacker sends the remainder of the attack. The internal router forwards the attack prefix and the target moves into a pre-failed state.

5.2.3 Discussion

With this process structure, we are able to identify more accurately the reasons for the specification violation. Firstly the timeout values are not equal and secondly the router delays a packet. However, there exist various general problems with these sorts of models:

1. We have many ways to place *tocks* and it may very well be that a particular placement covers timing based security gaps.

2. The synchronisation, especially in difficult systems, on the *tock* event between the different processes can cause deadlock. This particular problem is called *time stop*[1].

3. The other drawback about synchronising on *tock* is that it may force different timeout mechanisms to timeout at the same time. This would lead to an unintended timeout-synchronisation.

The recommended way of testing timing issues would then be first to use the sliding choice operator, and if this test fails one should use the discrete time model to obtain more expressive traces. However, using this approach, we have to be certain that whenever the first model does not find an attack then the second does not also. In the next section, we show that whenever the first model refines the specification then the second does also, and we will discuss a function that relives us from placing the *tock* events manually.

5.3 Towards a more complete analysis

In this Section, we prove that whenever there is a flaw in our discrete timeout model then this flaw also exists in the sliding choice model. Thus, we use the sliding choice timeout model in order to determine whether the discrete model harbors specification violations.

In the following proof, we use the prefixes T and U to distinguish between processes in the *tock* and the sliding choice based model.

In order to show that this holds, we will show the following: if there is an attack in tr on the discrete-time model, then $tr \backslash \{tock\}$ is an attack on the sliding choice model ($\triangleright$). Hence we have to show that:

$$TimedModel \backslash \{tock\} \sqsupseteq_T UntimedModel.$$

[1] A *time stop* takes place whenever: two processes are synchronised on a data channel and on the *tock* event, the receiver process can not engage in a *tock* event without receiving a certain value from the data channel, the value is provided by the sender process and the sending process however has to engage first in a *tock* event before he can transmit the value.

Where $TimedModel$ is using the *tock* event to measure the passage of time (as described in section 5.1) and $UntimedModel$ uses the sliding choice operator (as described in 5.2). We use the following law to isolate the problem:

$$(P \parallel Q) \setminus \{A\} \sqsupseteq_T (P \setminus A\}) \parallel (Q \setminus \{A\}).$$

This justifies the following refinement:

$$\begin{aligned}&(TAttacker(a) \parallel TRouter(a,b) \parallel TIDS(OS, buff, sigs, max, dist) \\ &\parallel TRouter(a,b) \parallel TTarget(OS, buff, sigs, max)) \setminus \{tock\}\end{aligned} \tag{5.1}$$

$$\sqsupseteq_T$$

$$\begin{aligned}&((TAttacker(a) \setminus \{tock\}) \parallel (TRouter(a,b) \setminus \{tock\}) \\ &\parallel (TIDS(OS, buff, sigs, max, dist) \setminus \{tock\}) \parallel (TRouter(a,b) \setminus \{tock\}) \\ &\parallel (TTarget(OS, buff, sigs, max) \setminus \{tock\})).\end{aligned}$$

If $(P \parallel Q)$ and $(P' \parallel Q')$ and $(P \sqsubseteq_T P')$ and $(Q \sqsubseteq_T Q')$ hold then $(P \parallel Q) \sqsubseteq_T (P' \parallel Q')$ (by monotonicity). Therefore, we only have to show that every component in the *tock* model refines its counterpart in the sliding choice model.

$$\begin{aligned}&TIDS(OS, buff, sigs, max, dist, timeout) \setminus \{tock\} \\ &\sqsupseteq_T UIDS(OS, buff, sigs, max, dist) \\ &\wedge \\ &TTarget(OS, buff, sigs, max, dist, timeout) \setminus \{tock\} \\ &\sqsupseteq_T UTarget(OS, buff, sigs, max, dist) \\ &\wedge \\ &TRouter(in, out) \setminus \{tock\} \\ &\sqsupseteq_T URouter(in, out) \\ &\wedge \\ &TAttacker \setminus \{tock\} \\ &\sqsupseteq_T UAttacker.\end{aligned}$$

It is easy to see that this holds for the attacker and the router process. Therefore, the remainder of our problem is to show that:

$$\begin{aligned}&TIDS(OS, buff, sigs, max, dist, timeout) \setminus \{tock\} \\ &\sqsupseteq_T UIDS(OS, buff, sigs, max, dist) \\ &\wedge \\ &TTarget(OS, buff, sigs, max, dist, timeout) \setminus \{tock\} \\ &\sqsupseteq_T UTarget(OS, buff, sigs, max, dist).\end{aligned}$$

Since the structure of the target and the IDS process is very similar we will only elaborate on the target process. Let us focus on $TResetTarget \setminus \{tock\}$. Firstly

we push the hiding through:

$$
\begin{array}{l}
TResetTarget(OS, buff, sigs, max, timeout, tocks) \setminus \{tock\} = \\
\quad TTarget(OS, buff, sigs, max, timeout) \setminus \{tock\} \\
\quad \rhd \\
\quad \text{if } tocks = 1 \\
\qquad \text{then } flushTarget \rightarrow \\
\qquad\qquad TTarget(OS, \langle N, N, N, N, N \rangle, sigs, 0, timeout) \setminus \{tock\} \\
\qquad \text{else } TResetTarget(OS, buff, sigs, max, timeout, tocks - 1) \setminus \{tock\}.
\end{array}
$$

For distinguishability let us add an H, to represent hide, at the end of the process descriptor. Therefore we get $TResetTargetH$. We can use the UFP rule for showing that:

$$
\begin{array}{l}
TTargetH(OS, buff, sigs, max) =_T UTarget(OS, buff, sigs, max) \\
\wedge \\
TTarget2H(OS, buff, sigs, max, mf, fo2, data, buff2, timeout) \\
=_T \\
UTarget2(OS, buff, sigs, max, mf, fo2, data, buff2, timeout) \\
\wedge \\
TResetTargetH(OS, buff, sigs, max, timeout, tocks) \\
=_T \\
UResetTarget(OS, buff, sigs, max, timeout, tocks).
\end{array}
$$

To do so we define a vector $\underline{X}$. Each element of $\underline{X}$ is additionally parameterised by whether it is in state 1, 2 or 3. To define the fixed point mapping corresponding to this recursion, given $\underline{X}$, we write $X(1, OS, buff, sigs, max, dist)$, $X(2, OS, buff, sigs, max, mf, fragmentoffset, data, buff2, timeout)$ and $X(3, OS, buff, sigs, max, timeout, tocks)$ for $TTargetH$, $TTarget2H$ and $TResetTargetH$ respectively. The following fixed point mapping defines $TTargetH$:

$$
\begin{array}{l}
\underline{F}(\underline{X})(1, OS, buff, sigs, max, timeout) = \\
\quad in?mf?fo2?ttl?data \rightarrow \\
\quad \text{let } b1 = overwrite(buff, fo2, data) \\
\quad \text{within } X(2, OS, buff, sigs, max, mf, fo2, data, b1, timeout).
\end{array}
$$

The following fixed point mapping defines $TTarget2H$:

$$\begin{array}{l}
\underline{F}(\underline{X})(2, OS, buff, sigs, max, 0, fo2, data, b1, timeout) = \\
\quad \text{if } nth(buff, fo2)! = N \wedge OS = 0 \\
\quad \text{then } X(3, OS, buff, sigs, max, timeout, timeout) \\
\quad \text{else if } allFilled(b1, fo2) \\
\qquad \text{then if } check(b1, sigs, fo2) \\
\qquad\quad \text{then } fail \rightarrow STOP \\
\qquad\quad \text{else } X(1, OS, \langle N, N, N, N, N\rangle, sigs, 0, timeout) \\
\qquad \text{else } X(3, OS, b1, sigs, fo2, timeout, timeout).
\end{array}$$

$$\begin{array}{l}
\underline{F}(\underline{X})(2, OS, buff, sigs, max, 1, fo2, data, b1, timeout) = \\
\quad \text{if } nth(buff, fo2) \neq N \\
\quad \text{then if } OS = 0 \\
\qquad \text{then } X(3, OS, buff, sigs, max, timeout, timeout)) \\
\qquad \text{else } X(3, OS, b1, sigs, max, timeout, timeout) \\
\quad \text{else if } allFilled(b1, max) \wedge max \neq 0 \\
\qquad \text{then if } check(b1, sigs, max) \\
\qquad\quad \text{then } fail \rightarrow STOP \\
\qquad\quad \text{else } X(1, OS, \langle N, N, N, N, N\rangle, sigs, 0, timeout) \\
\qquad \text{else } X(3, OS, b1, sigs, max, timeout, timeout).
\end{array}$$

The following fixed point mapping defines $TResetTargetH$:

$$\begin{array}{l}
\underline{F}(\underline{X})(3, OS, buff, sigs, max, timeout, tocks) = \\
\quad X(1, OS, buff, sigs, max, timeout) \\
\quad \triangleright \\
\quad \text{if } tocks = 1 \\
\qquad \text{then } flushTarget \rightarrow X(1, OS, \langle N, N, N, N, N\rangle, sigs, 0, timeout) \\
\qquad \text{else } X(3, OS, buff, sigs, max, timeout, tocks - 1).
\end{array}$$

We also have to define a vector $\underline{Y}$ that represents the un-timed processes. Therefore we get the following allocation.

$$\begin{array}{l}
Y(1, OS, buff, sigs, max) = UTarget(OS, buff, sigs, max) \\
Y(2, OS, buff, sigs, max, mf, fo2, data, buff2) = \\
\quad UTarget2(OS, buff, sigs, max, mf, fo2, data, buff2) \\
Y(3, OS, buff, sigs, max) = UResetTarget(OS, buff, sigs, max).
\end{array}$$

Now we have to show that $\underline{F}(\underline{Y}) = \underline{Y}$.

State 1

$$\underline{F}(\underline{Y})(1, OS, \mathit{buff}, \mathit{sigs}, \mathit{max})$$

$$
\begin{aligned}
&= (\mathit{by\ definition\ of\ } \underline{\mathrm{F}}) \\
&\quad \mathit{in?mf?fo2?ttl?data} \rightarrow \\
&\quad \mathrm{let}\ b1 = \mathit{overwrite}(\mathit{buff}, \mathit{fo2}, \mathit{data}) \\
&\quad \mathrm{within}\ Y(2, OS, \mathit{buff}, \mathit{sigs}, \mathit{max}, \mathit{mf}, \mathit{fo2}, \mathit{data}, b1) \\
&= (\mathit{by\ definion\ of\ } \underline{\mathrm{Y}}) \\
&\quad \mathit{in?mf?fo2?ttl?data} \rightarrow \\
&\quad \mathrm{let}\ b1 = \mathit{overwrite}(\mathit{buff}, \mathit{fo2}, \mathit{data}) \\
&\quad \mathrm{within}\ \mathit{UTarget2}(OS, \mathit{buff}, \mathit{sigs}, \mathit{max}, \mathit{mf}, \mathit{fo2}, \mathit{data}, \mathit{buff2}) \\
&= (\mathit{by\ definion\ of\ UTarget}) \\
&\quad \mathit{UTarget}(OS, \mathit{buff}, \mathit{sigs}, \mathit{max}) \\
\\
&= (\mathit{by\ definion\ of\ } \underline{\mathrm{Y}}) \\
&\quad Y(1, OS, \mathit{buff}, \mathit{sigs}, \mathit{max})
\end{aligned}
$$

State 2

$$
\begin{aligned}
&\underline{F}(\underline{Y})(2, OS, \mathit{buff}, \mathit{sigs}, \mathit{max}, \mathit{mf}, \mathit{fo2}, \mathit{data}, \mathit{buff2}) \\
&= (\mathit{by\ definition\ of\ } \underline{F}) \\
&\quad \ldots \\
&= (\mathit{by\ definion\ of\ } \underline{Y}) \\
&\quad \ldots \\
&= (\mathit{by\ definion\ of\ UTarget2}) \\
&\quad \ldots \\
&= (\mathit{by\ definion\ of\ } \underline{Y}) \\
&\quad \ldots \\
&\quad Y(2, OS, \mathit{buff}, \mathit{sigs}, \mathit{max}, \mathit{mf}, \mathit{fo2}, \mathit{data}, \mathit{buff2})
\end{aligned}
$$

Since the succession of transformations is equal to Stage 1 we will not elaborate on them. It is straightforward to see that this holds as well.

State 3

$$\underline{F}(\underline{Y})(3, OS, \mathit{buff}, \mathit{sigs}, \mathit{max})$$

$$
\begin{aligned}
&= (\mathit{by\ definition\ of\ } \underline{F}) \\
&\quad Y(1, OS, \mathit{buff}, \mathit{sigs}, \mathit{max}) \\
&\quad \triangleright \\
&\quad \mathrm{if}\ \mathit{tocks} = 1 \\
&\qquad \mathrm{then}\ \mathit{flushTarget} \rightarrow Y(1, OS, \langle N, N, N, N, N \rangle, \mathit{sigs}, 0) \\
&\qquad \mathrm{else}\ Y(3, OS, \mathit{buff}, \mathit{sigs}, \mathit{max}) \\
\\
&= (\mathit{by\ definion\ of\ } \underline{Y})
\end{aligned}
$$

$UTarget(OS, buff, sigs, max)$
$\triangleright$
if $tocks = 1$
then $flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N \rangle, sigs, 0)$
else $UResetTarget(OS, buff, sigs, max)$

$=$ ($using\ the\ distribution\ law$)
if $tocks = 1$
then $UTarget(OS, buff, sigs, max)$
$\triangleright flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N \rangle, sigs, 0)$
else $UTarget(OS, buff, sigs, max)$
$\triangleright UResetTarget(OS, buff, sigs, max)$

$=$ ($by\ unwinding\ UResetTarget$)
if $tocks = 1$
then $UTarget(OS, buff, sigs, max)$
$\triangleright flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N \rangle, sigs, 0)$
else $UTarget(OS, buff, sigs, max) \triangleright$
$(UTarget(OS, buff, sigs, max)$
$\triangleright flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N \rangle, sigs, 0))$

$=$ ($by\ using\ following\ steps$
$P \triangleright Q$
$=_T \quad (P \sqcap STOP) \,\Box\, Q$
$=_T \quad P \,\Box\, Q$
$we\ get$)
if $tocks = 1$
then $UTarget(OS, buff, sigs, max)$
$\Box\ flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N \rangle, sigs, 0)$
else $UTarget(OS, buff, sigs, max)\ \Box$
$(UTarget(OS, buff, sigs, max)$
$\Box\ flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N \rangle, sigs, 0))$

$=$ ($distribution\ law$)
if $tocks = 1$
then $UTarget(OS, buff, sigs, max)$
$\Box\ flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N \rangle, sigs, 0)$
else $UTarget(OS, buff, sigs, max)$
$\Box\ flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N \rangle, sigs, 0)$

$$
\begin{array}{l}
= (since\ (\text{if}\ \ b\ \ \text{then}\ \ P\ \ \text{else}\ \ Q)\ \sqsupseteq_T\ \ P\ \ \Box\ \ Q\ \ we\ get) \\
\quad (UTarget(OS, buff, sigs, max) \\
\qquad \Box\ flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N\rangle, sigs, 0)) \\
\quad \Box \quad (UTarget(OS, buff, sigs, max) \\
\qquad \Box\ flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N\rangle, sigs, 0))
\end{array}
$$

$$
\begin{array}{l}
= (distribution\ law) \\
\quad (UTarget(OS, buff, sigs, max) \\
\qquad \Box\ flushTarget \rightarrow UTarget(OS, \langle N, N, N, N, N\rangle, sigs, 0))
\end{array}
$$

$$
\begin{array}{l}
= (by\ definion\ of\ UResetTarget) \\
\quad UResetTarget(OS, buff, sigs, max)
\end{array}
$$

$$
\begin{array}{l}
= (by\ definion\ of\ \underline{Y}) \\
Y(3, OS, buff, sigs, max).
\end{array}
$$

Finally, we can conclude that $TTarget$ refines $UTarget$. Hence, condition (5.1) holds, showing that our conjecture, that whenever a specification violating trace tr exists in our timed model, then $tr \setminus \{tock\}$ exists in the untimed, holds true. To exercise this kind of proof for every timeout model seems to be impractical. We have to find a way to establish this relationship between the untimed and timed timeout processes more easily. One way to do so is by generalising our result. Ideally, this should also relieve us from the tedious process of placing the *tock* events on the right places.

Chapter 6

Generalisation

From this very specific result, we can extrapolate further — to a general case. Assuming f is a function that converts a *tock*–free (untimed) process into a timed–deadlock–free process, by adding a *tock*–loop to every stable state, inserting the *tock* event into its synchronisation sets and by replacing every $TimeoutUT(P,Q)$ by $TimeoutT(P,Q,T)$ for some T — then

$$P \sqsubseteq_T f(P) \setminus \{tock\} \text{ or}$$
$$P \sqsubseteq_T P_{Timed} \setminus \{tock\}.$$

holds for all *tock*–free processes P. Thus, in the traces model and using untimed safety specifications, it is enough to verify the process with the simpler timeout version.

First we have to define a relabelling function g_x that allows us to define a proper *tock* based timeout. Note this function and the general structure of our timeout process is similar to the one given in [Oua01, Sch00a].

Let $\Sigma_P = \{P.x | x \in \Sigma\}$ and $\Sigma_Q = \{Q.x | x \in \Sigma\}$. The functions g_P and g_Q are of type $\Sigma_{tock} \cup \Sigma_P \cup \Sigma_Q \longrightarrow \Sigma_{tock} \cup \Sigma_P \cup \Sigma_Q$ and are defined as:

$$g_x(y) = \text{if } y \in \Sigma \text{ then } x.y \text{ else } y.$$

The function $TimeoutUT(P,Q)$ and $TimeoutT(P,Q,T)$ are defined as follows:

$$TimeoutUT(P,Q) = P \rhd Q.$$

and

$$\begin{aligned} &TimeoutT(P,Q,T) = \\ &\quad g_P^{-1}(g_Q^{-1}((g_P(P) \underset{\{tock\}}{\parallel} g_Q(R(Q,T))) \\ &\qquad \underset{\Sigma_P \cup \Sigma_Q}{\parallel} (Run_{\Sigma_P} \,\Box\, Run_{\Sigma_Q}))) \setminus \{timeout\}. \end{aligned}$$

where:

$$
\begin{aligned}
&Run_X = \\
&\quad x : X \rightarrow Run_X \\
\\
&R(Q,T) = \\
&\quad Wait(T)\ ;\ timeout \rightarrow Q \\
\\
&Wait(t) = \\
&\quad \text{if } t = 0 \\
&\qquad \text{then } SKIP \\
&\qquad \text{else } tock \rightarrow Wait(t-1).
\end{aligned}
$$

The event *timeout* is used as a guard and must never appear in the regular alphabet of the involved processes. The function that converts the sliding choice timeout model into a *tock* based timeout model is called f and is defined as:

$$f(TimeoutUT(P,Q)) = TimeoutT(f(P), f(Q), T)\ for\ some\ T$$

$$f(STOP) = \mu\, z.tock \rightarrow z$$

$$f(a \rightarrow P) = \mu\, z.a \rightarrow f(P) \,\Box\, tock \rightarrow z$$

$$
\begin{aligned}
&f(P \,\Box\, Q) = \\
&\quad g^{-1}_{f(P)}(g^{-1}_{f(Q)}((g_{f(P)}(f(P))) \underset{\{tock\}}{\|} g_{f(Q)}(f(Q)) \\
&\qquad\qquad \underset{\Sigma_{f(P)} \cup \Sigma_{f(Q)}}{\|} (Run_{\Sigma_{f(P)}} \,\Box\, Run_{\Sigma_{f(Q)}})))
\end{aligned}
$$

$$f(P \sqcap Q) = f(P) \sqcap f(Q)$$

$$f(P \underset{Y}{\|} Q) = f(P) \underset{Y \cup \{tock\}}{\|} f(Q).$$

All the conversions seem to be intuitive except the conversion of the external choice operator. If we would simply add a *tock* loop to the external choice, as in the other cases, the choice would become non-deterministic. Since $f(P)$ and $f(Q)$ can engage in *tock* events, we would have the *choice* between three *tock* events. Only one of them representing the intended *tock* event that would return to the initial state. Hence, we have to make certain that both $f(P)$ and $f(Q)$ are able to engage in tocks and that the external choice remains unresolved [Oua01].

Now we have to show that our conjecture ($f(P) \setminus \{tock\} \sqsupseteq_T P$) holds for all P. We do so by showing that our claim holds for every necessary CSP operator,

starting with the conversion of $\triangleright$:

$$
\begin{aligned}
& f(P \triangleright Q) \setminus \{tock\} \\
= & \ (by\ definition\ of\ TimeoutUT) \\
& f(TimeoutUT(P,Q)) \setminus \{tock\} \\
= & \ (by\ definition\ of\ f()) \\
& TimeoutT(f(P), f(Q), T) \\
= & \ (by\ definition\ of\ TimeoutT) \\
& g^{-1}_{f(P)}(g^{-1}_{f(Q)}((g_{f(P)}(f(P))) \underset{\{tock\}}{\parallel} g_{f(Q)}(R(f(Q),T)) \\
& \qquad \underset{\Sigma_{f(P)} \cup \Sigma_{f(Q)}}{\parallel} (Run_{\Sigma_{f(P)}} \Box Run_{\Sigma_{f(Q)}}))) \setminus \{timeout\} \setminus \{tock\} \\
& and\ R(f(Q),T) = Wait(T)\ ;\ timeout \rightarrow f(Q) \\
= & \ (partially\ resolving\ \setminus \{tock\}) \\
& g^{-1}_{f(P)}(g^{-1}_{f(Q)}(((g_{f(P)}(f(P) \setminus \{tock\}))) \underset{\{tock\}}{\parallel} (g_{f(Q)}(R(f(Q),T)) \setminus \{tock\}) \\
& \qquad \underset{\Sigma_{f(P)} \cup \Sigma_{f(Q)}}{\parallel} (Run_{\Sigma_{f(P)}} \Box Run_{\Sigma_{f(Q)}}))) \setminus \{timeout\} \\
= & \ (resolving\ g_{f(Q)}(R(f(Q),T)) \setminus \{tock\}) \\
& g_{f(Q)}(Wait(T)\ ;\ timeout \rightarrow f(Q)) \setminus \{tock\} \\
= & \ (using\ Wait(t) \setminus \{tock\} =_T SKIP\ and\ SKIP\ ; timeout \rightarrow f(Q) =_T timeout \rightarrow f(Q)) \\
& g^{-1}_{f(P)}(g^{-1}_{f(Q)}(((g_{f(P)}(f(P) \setminus \{tock\}))) \underset{\{tock\}}{\parallel} (g_{f(Q)}(timeout \rightarrow (f(Q) \setminus \{tock\}))) \\
& \qquad \underset{\Sigma_{f(P)} \cup \Sigma_{f(Q)}}{\parallel} (Run_{\Sigma_{f(P)}} \Box Run_{\Sigma_{f(Q)}}))) \setminus \{timeout\} \\
= & \ (using\ P \setminus \{tock\} \underset{\{tock\}}{\parallel} Q \setminus \{tock\} \sqsupseteq_T P \ ||| \ Q) \\
= & \ (using\ Z = (P \ ||| Q) \underset{\Sigma_P \cup \Sigma_Q}{\parallel} (Run_{\Sigma_P} \Box Run_{\Sigma_Q})) \\
= & \ (using\ Z =_T ((P \underset{\Sigma_P}{\parallel} Run_{\Sigma_P}) \Box (Q \underset{\Sigma_Q}{\parallel} Run_{\Sigma_Q})) =_T P \Box Q) \\
& g^{-1}_{f(P)}(g^{-1}_{f(Q)}((g_{f(P)}(f(P) \setminus \{tock\})) \Box (g_{f(Q)}(timeout \rightarrow (f(Q) \setminus \{tock\})))))) \setminus \{timeout\} \\
= & \ (apply\ g^{-1}_X(g_X(X)) = X\ and\ g^{-1}_X(g_Y(Y)) = g_Y(Y)) \\
& (f(P) \setminus \{tock\}) \Box (f(Q) \setminus \{tock\}) \\
= & \ (using\ P \Box Q =_T (P \sqcap STOP) \Box Q =_T P \triangleright Q) \\
& (f(P) \setminus \{tock\}) \triangleright (f(Q) \setminus \{tock\}) \\
\sqsupseteq_T & \ (by\ structural\ inductive\ hypothesis) \\
& P \triangleright Q.
\end{aligned}
$$

The remaining part of the proof verifies whether the same holds for the other necessary CSP operators.
Case $f(STOP)$:

$$
\begin{aligned}
& f(STOP) \setminus \{tock\} = \\
& (\mu\, z.tock \rightarrow z) \setminus \{tock\} \\
= & \ (\mu\, z.z) \\
& div \\
\sqsupseteq_T & \ (in\ the\ traces\ model\ this\ is\ equivalent\ to) \\
& STOP.
\end{aligned}
$$

Case $f(a \rightarrow P)$:

$$
\begin{aligned}
& f(a \rightarrow P) \setminus \{tock\} = \\
& (\mu\, z.a \rightarrow f(P) \,\Box\, tock \rightarrow z) \setminus \{tock\} \\
= & \ (resolving\ the\ hide\ operator) \\
& \mu\, z.a \rightarrow (f(P) \setminus \{tock\}) \rhd z \\
= & \ (in\ the\ traces\ model\ this\ is\ equivalent\ to) \\
& a \rightarrow (f(P) \setminus \{tock\}) \\
\sqsupseteq_T & \ (by\ structural\ inductive\ hypothesis) \\
& a \rightarrow P.
\end{aligned}
$$

Case $f(P \,\Box\, Q)$:

$$
\begin{aligned}
& f(P \,\Box\, Q) \setminus \{tock\} = \\
= & \ (by\ definition\ of\ f()) \\
& g^{-1}_{f(P)}(g^{-1}_{f(Q)}((g_{f(P)}(f(P))) \underset{\{tock\}}{\parallel} g_{f(Q)}(f(Q)) \\
& \qquad \underset{\Sigma_{f(P)} \cup \Sigma_{f(Q)}}{\parallel} (Run_{\Sigma_{f(P)}} \,\Box\, Run_{\Sigma_{f(Q)}}))) \setminus \{tock\} \\
= & \ (partially\ resolving\ \setminus \{tock\}) \\
& g^{-1}_{f(P)}(g^{-1}_{f(Q)}(((g_{f(P)}(f(P) \setminus \{tock\}))) \underset{\{tock\}}{\parallel} (g_{f(Q)}(f(Q)) \setminus \{tock\}) \\
& \qquad \underset{\Sigma_{f(P)} \cup \Sigma_{f(Q)}}{\parallel} (Run_{\Sigma_{f(P)}} \,\Box\, Run_{\Sigma_{f(Q)}}))) \\
= & \ (using\ P \setminus \{tock\} \underset{\{tock\}}{\parallel} Q \setminus \{tock\} \sqsupseteq_T P \,|||\, Q) \\
= & \ (using\ Z = (P \,|||Q) \underset{\Sigma_P \cup \Sigma_Q}{\parallel} (Run_{\Sigma_P} \,\Box\, Run_{\Sigma_Q})) \\
= & \ (using\ Z =_T ((P \underset{\Sigma_P}{\parallel} Run_{\Sigma_P}) \,\Box\, (Q \underset{\Sigma_Q}{\parallel} Run_{\Sigma_Q})) =_T P \,\Box\, Q) \\
& g^{-1}_{f(P)}(g^{-1}_{f(Q)}(((g_{f(P)}(f(P) \setminus \{tock\})) \,\Box\, (g_{f(Q)}(f(Q) \setminus \{tock\}))))) \\
= & \ (apply\ g^{-1}_X(g_X(X)) = X\ and\ g^{-1}_X(g_Y(Y)) = g_Y(Y)) \\
& (f(P) \setminus \{tock\}) \,\Box\, (f(Q) \setminus \{tock\}) \\
\sqsupseteq_T & \ (by\ structural\ inductive\ hypothesis) \\
& P \,\Box\, Q.
\end{aligned}
$$

Case $f(P \sqcap Q)$:

$$
\begin{aligned}
& f(P \sqcap Q) \setminus \{tock\} = \\
& (f(P) \sqcap f(Q)) \setminus \{tock\} \\
= & \ (by\ resolving\ the\ hide\ operator) \\
& (f(P) \setminus \{tock\}) \sqcap (f(Q) \setminus \{tock\}) \\
\sqsupseteq_T & \ (by\ structural\ inductive\ hypothesis) \\
& P \sqcap Q.
\end{aligned}
$$

Case $f(P \underset{Y}{\parallel} Q)$:

$$
\begin{aligned}
& f(P \underset{Y}{\parallel} Q) \setminus \{tock\} = \\
& (f(P) \underset{Y \cup \{tock\}}{\parallel} f(Q)) \setminus \{tock\} \\
= & \ (applying\ f()) \\
& (f(P) \underset{Y \cup \{tock\}}{\parallel} f(Q)) \setminus \{tock\} \\
\sqsupseteq_T & \ (by\ structural\ inductive\ hypothesis) \\
& P \underset{Y}{\parallel} Q.
\end{aligned}
$$

After establishing that this is true for every used operator, we can conclude that our conjecture $P \sqsubseteq_T P_{Timed} \setminus \{tock\}$ holds for every process that uses our type of timeout and the operators we tested.

6.1 Conclusion

In this Chapter we presented different ways to enhance our models to cover time and derived a method to reason about a more general case.

Firstly, we used the sliding choice operator to determine whether or not an attack exists. The problem with the resulting trace was its ambiguity: one could not be certain were the vulnerability originated from. It could have been an internal router that delayed the packet forwarding or different timeout values used by the target and IDS. To obtain more precise counter-examples, we introduced the event *tock* that symbolises the passage of time. Whilst the resulting discrete timeout model was more complicated, the traces generated by FDR were easy to interpret.

We did not suggest an improved version for these models, since the datagram always reaches the IDS or the target first. So even if the timeout values are equal, the attacker still has the possibility to succeed. However, this kind of attack is unlikely in the real world, since the attacker has to predict the exact timing behaviour of all components that lie between the IDS and the target.

This examination finished with a discussion about the relation that exists between a *tock* and sliding choice based model. We showed that whenever the untimed model does not find an attack the discrete-time model is equally free from flaws. However, this specific result is unsatisfactory, since for every change, in the structure of the participating processes, we have to show that this relationship holds again.

Hence, we generalised our models to find a more abstract way to establish this relationship. For that a function was derived that converts every untimed process using a sliding choice operator into a process that uses discrete time. This was followed by a proof that sets our claim on formal grounds. The result of our proof was that, for untimed safety specs, we only have to verify the untimed processes

without fearing to miss a specification violation in the discrete timeout model.

This differs from [Oua01] since he mainly investigates the relationship between discrete and continuous TCSP[1]. In contrast to [Oua01], [Sch97] is concerned with the mapping of untimed onto continuous timed processes. We on the other hand, present a way how one can lift a untimed CSP model to a discrete-timed CSP model. Additionally, we only investigated the basic operators of CSP.

Returning to the practical consequences, we have shown that by omitting timing issues a blinding possibility in the TTL model remains undiscovered. This leads us to the general problem, which we have mentioned before, that we can never be certain whether our simplification of real world features within a model introduces a false sense of security. This could be an avenue for future work (more on that topic in chapter 12.3).

[1] In order to use FDR he also describes a function that projects TCSP onto a *tock* based untimed CSP model.

Chapter 7

Trusted Computing Architectures

This chapter introduces the abstract concept of trusted computing. It will briefly discuss the current approaches that are suggested by Microsoft, TCG and Intel. The focus of the introduction will be on the TCPA solution. It will describe scope, design features and the various definitions of trust that are related to this architecture. This is followed by a more detailed overview of the different subsections of the TCPA model. In particular the internal trust relationships that are required for proper operation, integrity verification and reporting, creation of trusted identities and the protected storage. Since the architecture uses various protocols that have to be verified, we will also introduce *Casper* as a protocol verification tool that uses an easy to read and write message description language and converts it into a CSP_m script. This chapter concludes with a summary and a brief discussion of the relationships between the various trusted computing concepts that were introduced earlier.

7.1 The directive of TCPA

In times of e-commerce and upcoming e-governance it is essential that participants can rely on their platforms and on the computing devices with which they are communicating. This trust includes the secure communication, storage and use of sensitive data. Several surveys have shown that lack of trust is the main issue for the limited participation in e-commerce [ATT99, Che99].

In January 1999, Compaq, HP, IBM, Intel and Microsoft founded an organisation called Trusted Computing Platform Alliance (TCPA). The main intention was to stimulate participation and discussion about a hardware-software approach to make today's computers more secure [TCPA02, TCPA03d, TCG05]. This approach transfers functionality that is essential for the security of the overall system into hardware components. In April 2003 the steering committee which

is comprised of the founders of TCPA decided to define a new group called the Trusted Computing Group (TCG)[Hei03, Pri03], due to problems of finding a consensus between all members in the TCPA. In the following pages we will use the label TCPA, since the press and the specifications refer to the mechanisms described below as TCPA mechanisms.

Software-only solutions are weak because they depend on the correct installation of their trusted components and software. Additionally, the execution of applications can freely interfere with other software processes. All these facts lead back to the base problem that software can not vouch for its own integrity [LABW92].

TCPA aims to build and standardise a solid foundation to overcome this problem by including hardware in their approach. The TCPA itself describes its objective as:

> through the collaboration of platform, software, and technology vendors, [to] develop a specification that delivers an enhanced HW- OS-based trusted computing platform that enhances customers' trusted domains [TCPA00, Pea02].

The Alliance's approach is twofold. First, they specify hardware, operating systems and additional software so that the required hardware can be produced for the mainstream market. To succeed on the mainstream market the product has to be of little cost. As we will see later on, this paradigm determines many features of the TCPA core, called the Trusted Platform Module (TPM). Second, they animate vendors to support the features of the new hardware.

7.1.1 Scope

Figure 7.1 depicts the scope of the TCPA specification. The specification itself deals with the hardware, OS, and BIOS level. Moreover, it uses complementary technologies such as Transport Layer Security (TLS), S-MIME, smart cards, X.509, IPSEC, VPN, PKI and IKE.

7.1.2 Design features

The Alliance defines a certain set of capabilities that should provide the user with trust.

1. The TCPA hardware must support cryptographic primitives such as RSA. Bulk or symmetric encryption is not included. It would be too expensive to provide a mechanism that would not lead into a bottleneck for the whole system[1]; For such heavily taxing operations the CPU is used.

[1] At this point the TCPA specification versions 1.1b and 1.2 vary. 1.1b does not include symmetric encryption whereas 1.2 supports Advanced Encryption Standard (AES) [NIST01].

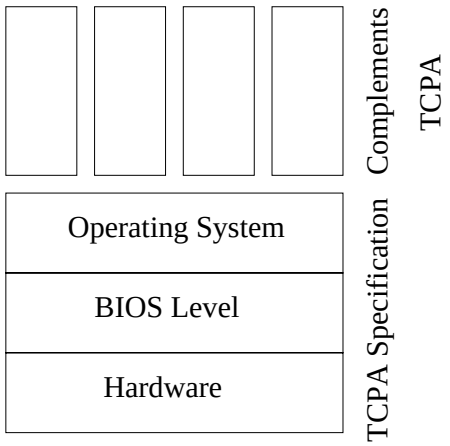

Figure 7.1: Scope of the TCPA

2. It must include a mechanism that allows the user to protect his privacy. There should be no global secret that would allow a successful penetrator to get more knowledge than the information about the trusted platform he has successfully hijacked.

3. It should provide a low cost protected environment.

4. It should provide ubiquitous security.

The last two principles are closely related. Both are concerned with the total-cost-of-ownership of the chip. If the implementation would be expensive, then it would be rather unlikely that it would become available on the mass market — which renders this technology not ubiquitous.

7.1.3 Definition of trust

The exact definition of trust is difficult since trust has different properties and meanings — depending on the context in which it is used. Trust is not always transitive. If entity A trusts entity B and B in turn trusts C; then the fundamental issue is whether A trusts C. Depending on the context, this answer can be yes or no.

The trust relations that exist in a system may or may not be everlasting, since we may trust an entity at one point and after a certain period of time this entity may become un-reliable. We will elaborate on this point in our boot-sequence investigation. The same holds for the degree of trust we offer another entity; Summarising this, we can say:

> Trust is a psychological state comprising the intention to accept vulnerability based upon positive expectations of the intentions or behaviour of another [RSBC09].

The trusted platform claims to achieve this trust by logging executed actions and by supplying the user with evidence of these actions.

7.1.4 Usage Scenarios

The TCPA gives many examples of how one can use their technology to improve daily life. In the following section we will describe several of these.

1. It is an inexpensive way of certifying that the platform is working as expected. The Alliance sets this in contrast to the secure platforms that are very expensive to certify and maintain. The TCPA architecture offers a mechanism for preserved information and platform integrity.

2. Remote users can obtain reliable information about the current state of the platform. The user can only prevent outsiders from getting the state information; however, if he chooses to inform them the information is reliable and correct. Hence, business partners can see whether the platform they are communicating with has been compromised.

3. This architecture allows the security community to implement more security features in general (e.g. protection against hostile main memory monitoring).

For the common user this means that he has access to stronger authorisation and authentication techniques. He can determine whether or not the platform is in a desired state, or whether it behaves in a desired manner. It even offers a sealed storage that allows the user to bind data not only to a specific software state but also to a particular platform. The user can create different IDs for different purposes, which allows him to have multiple non-related trust domains.

7.2 The trusted platform

Each trusted platform consists of a Trusted Platform System (TPS) and in figure 7.2 we see its architecture. Some readers may think of this as a Trusted Computing Block (TCB), since this definition has been around for some time now and is well-known. However, there are differences between these two approaches. The TCB includes all functions that are required to enforce the security policy of a system; these functions are formally assessed once in their lifetime. By contrast, the TPS does not offer functionality for bulk encryption or for maintaining access policies, but it is assessed continuously.

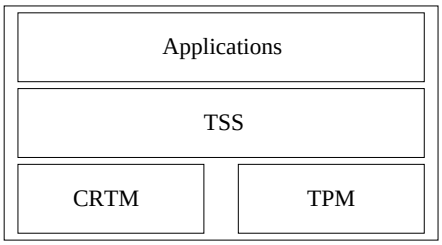

Figure 7.2: The Architecture of a TPS

The TPS itself is comprised of a Trusted Platform Module (TPM), a Core Root Trust Measurement (CRTM) and a Trusted Sub System (TSS). The TPM is an additional computing engine that contains all non-compromisable operations. The first piece of software that is executed to monitor and to self-validate the trusted platform is stored in the CRTM. The TSS represents the interface to software that is executed on the trusted platform or other external processes. These external processes can be other TPMs or simply the rest of the TPS.

Since authentication, trust and privacy are the main concerns of this initiative, it also includes a strong Certification Authority (CA) concept. These can be either regular CAs or so called Privacy Certification Authorities (P-CA). All CAs vouch for the genuineness of the trusted platform. The difference between the common CA and the P-CA is that the latter can be used to create additional Identities (ID). These IDs should make it practically impossible to link an ID's behavior to a specific person.

TCPA divides the logical structure of its architecture into three roots of trust: the Root of Trust for Measurement (RTM), the Root of Trust for Reporting (RTR) and the Root of Trust for Storing and reporting integrity metrics (RTS). The RTS is responsible for storing the monitored results and for storing protected data. It has to prevent modification and access of unauthorised users. The RTR is responsible for the generation of a hash sum of all logging-events. Because, we only consider the PC environment, we will not use this term further. The RTR is implemented in the TPM [TCPA02, TCPA03d].

The RTM is a — ideally — uncompromisable process that measures particular platform properties and stores the result in the measurement storage and a hash sum of the result in the TPM.

The RTM has the following properties:

1. it executes only code that is approved by the entity that is vouching for the RTM;

2. it conforms with the Protection Profile (PP) [TCPA03b];

3. it monitors the integrity metrics that keep track of the software environment;

4. it reports the results validly to the TPM;

5. it stores details of the monitoring process in a trusted platform measurement store.

The RTM represents the platform itself within the PC environment. To be protected against modifications and interference, the RTM code must be executed before anything else. The code must be trustworthy and be based on the KISS (Keep It Small And Simple) paradigm. Only if the code is kept as simple as possible can a proper evaluation guarantee faultlessness. This code is included in the Bios Boot Block (BBB) or in the BIOS instructions — this part is also called the CRTM (see above).

The other measurement agents are the OS loader and the OS itself. More generally, measurement agents are like the RTM; yet, they are not counted as a root of trust. Hence their integrity has to be measured before they can be used. The operating system contains a measurement agent to detect executions of processes that change the security related state of the platform. To prevent the log from corruption, a summary is stored within the TPM. In order to be a TCPA compliant OS it has to fulfil the following two points:

1. It must detect security related changes on the platform.

2. It must determine whether an event is worthy of logging or not. This includes a facility to manage this decision process. However, it should not be possible for a user to drop the logging activity below a certain amount, since then, in case of an integrity challenge by another remote entity, it would be impossible to determine whether or not the host is secure.

The TSS must be protected against software attacks. Thus it must be protected against local and remote interference. Even protection against hardware corruption is most desirable. The specification [TCPA03b, TCPA02] states that it is impossible to protect against a hardware-based attack that is initiated by an entity with large resources. [AK96] has shown that this indeed is true. Therefore, the Alliance confines itself to protecting against attacks that are practicable from entities with limited resources such as common users. Another topic along the lines of the latter is tamper-evidence [AK96]. We will discuss this in more depth in Section (Discussion or Conclusion of raw TPM analysis).

7.2.1 Relations within the TCPA architecture (Root of Trust)

As mentioned before, certain relations of trust are necessary to establish the TCPA trust model. In the following paragraph we will introduce these relations and their origins. After the TPM is generated, the Trusts Platform Module Entity (TPME) generates and signs the endorsement credential, which includes the

public key of the TPM endorsement key[2]. This certificate vouches for the uniqueness and genuineness of the TPM. The TPME is in most cases the manufacturer. At the next step a Validation Entity (VE) vouches for one or more parts linked to the subsystem. This certificate includes the integrity measurements which attest that the platform is working as desired. This certificate is also called the 'expected metrics'. The VE in most cases will be the manufacturer. The Conformance Entity (CE) generates the conformance credential which is required for attesting that the used design is compliant with the TCPA specification; this will be done by an independent evaluation centre. The Platform Entity (PE) signs and creates the platform certificate, which attests that this particular platform is using a valid TPM; and is therefore a genuine trusted platform. The last entity that is missing is the Privacy Certification Authority (P-CA), which links a genuine trusted platform to a certain ID. Only this P-CA has enough data to correlate all information about the user. Figure 7.3 shows the elements of the trusted platform and their corresponding certificates.

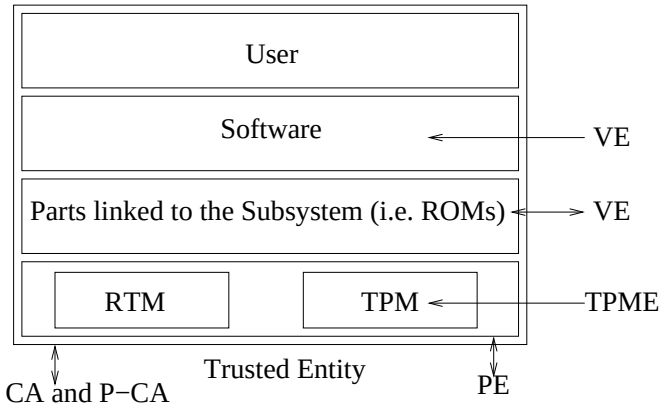

Figure 7.3: The Trust Relationship within TCPA

7.2.2 Creating a trusted identity for common interaction purposes

We distinguish between the following IDs: pseudonymous cryptographic ID, a TPM ID and an attestation ID (commonly of the form $(Label, PublicKey)_{P-CAsigned}$). Let's consider the case where the user has purchased a platform. The TPME, VE and CE have all certified that this platform can be trusted.

[2]The endorsement public key pair is generated once in the lifetime of a TPM (Specification 1.2 compliant devices have to include a mechanism to erase and re-create a new pair). This key pair uniquely identifies a particular TPM. The private part of the key is never exported. Thus, only the TPM itself can decrypt a message that was encrypted by the public endorsement key.

Participants of the ID generation protocol are the CA, the computer the user is using and the user himself. The basic process starts with the user sending information to the CA that vouches for being created on a genuine trusted platform. The information includes a signed certificate of the platform's manufacturer and uses a secret installed in the platform to show that it is unique and that it is indeed the platform for which the manufacturer is vouching. This secret is called the public part of the endorsement key and it is never shared with other arbitrary third parties. Instead, if the need arises, there are particular cryptographic attestation IDs that can be used.

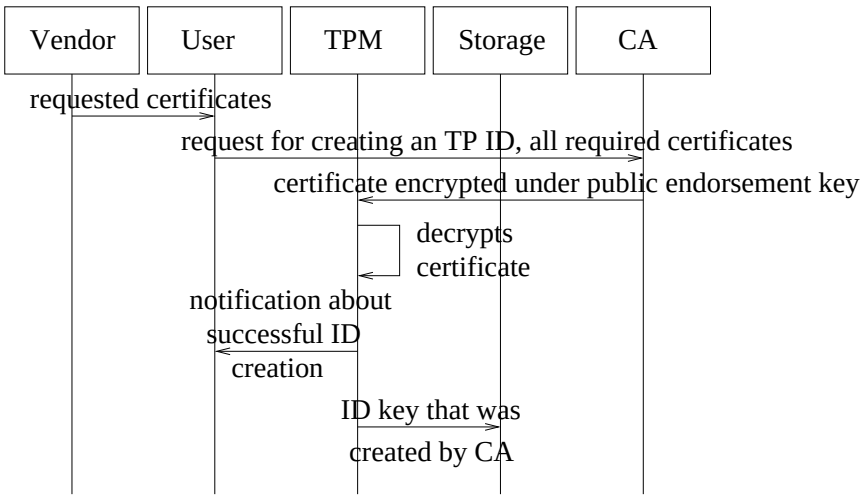

Figure 7.4: The generic creation of a Trusted Identity

The vendor provides all certificates that are required to attest that the trusted platform is genuine (see section 7.2.1). If the user decides to create another attestation identity he sends a request to a public certification authority that includes all required certificates. The CA generates an ID including a certificate. This data is encrypted with the public endorsement key of the TPM. Hence only the TPM can extract the data within the ID data. The TPM decrypts this data, verifies whether the response is correct and sends a notification to the user, which indicates that this particular identity is usable form now on. Finally, the ID key with all its data is stored securely in the protected storage. Note that this is only an abstract overview of the ID generation process. We deliberately omitted the complete transactions performed by the TPM. Furthermore, there are other ways to accomplish an ID generation; the interested reader is referred to [Pea02, TCPA02, TCPA03d]

Once this ID is successfully established, the user can create another ID called the pseudonymous cryptographic ID. The purpose of the latter is to have the freedom that no one can deduce the user's activities in the web. This PC-ID still

has the credibility of a trusted platform. However, it does not expose the public endorsement key at each transaction.

7.2.3 Integrity Verification and Reporting

The integrity verification and reporting process ensures that the chain of trust is established between the platform's software states. Before a process is executed, a measurement agent verifies whether the process will behave in the desired manner.

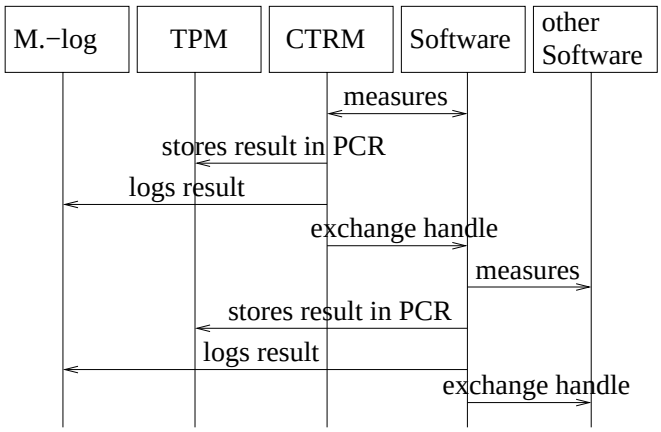

Figure 7.5: A generic Integrity verification Process

In the picture (7.5) we see the generic integrity verification and reporting process of a trusted platform. It starts with the execution of a bootstrapping process that measures the next software that has to be executed. The results are stored in the measurement log and the summary (the hash-sum) is used to update the process control registers (PCRs) in the TPM. Afterwards the execution handle is given to the evaluated code. This process does exactly the same, since it verifies the next execution layer, updates the log and the TPM, and passes the handle on to the verified software. Once the operating system is operational and an application sends a request to execute one of the TPM's commands, the regular audit function takes over. This audit function is very simplistic. For example, neither the parameters of executed commands nor the purpose of the commands are logged. There also exists an option to store predicted values that can force a trusted platform to stay always in a desired software state. Whenever a signature of an application deviates from these reference values, the trusted platform withdraws all the necessary resources (i.e. shuts the application). These signatures are provided by the vendor of the software and serve as reference material for a secure state. This data is not necessarily a digest: it can also

be a URL to the vendor's web site[3]. The supplier of this data vouches for its correctness by providing a certificate. They can be used in case of integrity challenges by remote users or by the authenticated boot mechanism.

7.2.4 Protected Storage

The TCPA architecture offers a facility to store data securely — the protected storage. This storage is theoretically infinitely extendable. It is not just able to link the data to a certain secret but also to a certain user or a particular application state on the trusted platform. The protected storage uses a tree structure to maintain its keys and data. In figure 7.6 we see this tree structure that starts with the Storage Root Key (SRK). This key is created once in the lifetime of the TPM. Because of its importance, the key should never be revealed to others. The SRK is used to encrypt its child blobs. It does not matter whether these blobs store data or keys. Blobs in common terminology stand for Binary Large Objects. In TCPA terms, blobs are binary large objects with a certain structure. We can distinguish between key and data blobs. Every blob is encrypted by a certain key. This key again is encrypted by its parent key.

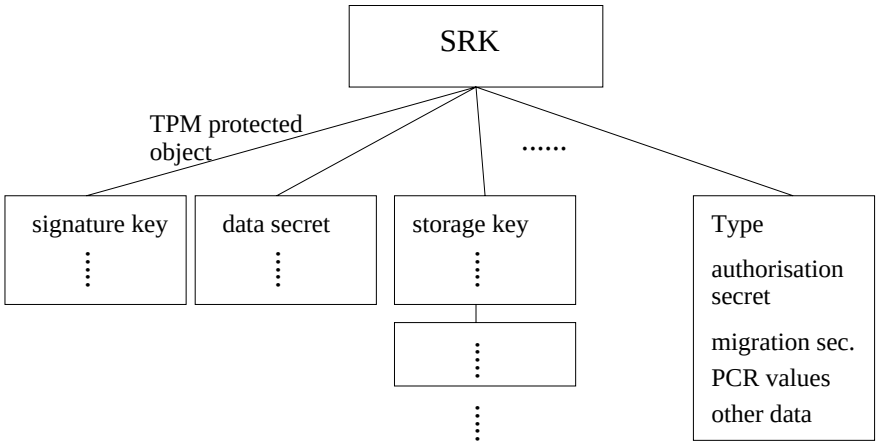

Figure 7.6: The Storage Hierarchy

The schema of an encryption is as follows: The user presents the authorisation secret for the key with which he wants to use to encrypt his data, the data and the desired software state of the platform. The TPM verifies the given information, encrypts the object with the requested key and places it as its child in the tree

[3]This contains a problem, as pointed out by Ross Anderson [And03], since this reference signature can be changed without the admission and knowledge of the trusted platform's owner. Hence, if the vendor decides to proscribe a particular application, he can do so by simply changing the reference values to a bogus value. These fake values, of course, will never be calculated by the TPM.

structure. Bulk encryption is delegated to the CPU. The unwrapping is the exact reverse process. After successful authentication for the used encryption key, the TPM unwraps the secret, extracts the desired state of the platform, checks the current state, and compares the two states. The overall key management is done by an external process, since it is believed that this functionality must not be trusted[4]. The tree itself has almost no restrictions except that signature keys and arbitrary data are always leaves.

About keys

There are two different types of keys, non-migratable and migratable. The first type can never be exported whereas the second type can be communicated to applications or even other platforms. In this fashion the user can link certain data to a particular host. The user has to encrypt his secret with a non-migratable key. Therefore, only this specific TPM is able to encrypt it. The $Asymmetric - Authorisation - Change - Protocol$ (AACP) makes use of this property. Signature keys are always non-migratable. Otherwise it would be possible to pretend that the origin of certain data elements is a genuine TPM when, in fact, it is not. Another subtlety is that identity keys are only signing data that originates from within the TPM.

7.2.5 The physical structure of the TPM

The TPM logically consists of protected capabilities and shielded locations. Protected capabilities are functions that must work in the desired way. Otherwise it would be impossible to detect that the platform behaves in an unexpected hostile manner. The shielded locations can store certain types of data. They should resist against modification and should prevent possible extraction of the data that is stored within the shielded locations. These locations are non-volatile and store only a few secrets, such as the endorsement and the storage root key. In contrast to the TPM, a cryptographic co-processor offers no functionality for integrity checks, for both data and processes, and offers no protected storage. The following parts are necessary to build a TPM:

The key generation component can generate RSA key pairs and symmetric keys of various lengths. It is essential that every TPM can generate its own keys, to make certain that no eavesdropping or extraction of keys is possible. This component can be further divided into the *RSA* generation part and the *Nonce* generation part. The *nonce* generator uses the *TPM random number generator* (see below) to create suitable values.

[4] In section 9 we investigate whether this claim truly holds.

The asymmetric encryption co-processor supports various calculations that are necessary for the asymmetric encryption. Basically it consists of an *RSA engine*, a *Signature engine* and a *Symmetric encryption engine*[5]. The *RSA engine* encrypts and the *Signature engine* signs data or certain values.

The computing engine processes and executes the TPM commands. In particular it manages various protocols, such as the Object Specific-Authorisation Protocol (OS-AP) and Object Independent-Authorisation Protocol (OI-AP) which we will analyse in chapter 8.

The HMAC generation ensures that the TPM can rely on the integrity of incoming and outgoing operands. The HMAC generation has to be according to RFC 2104 [KCB97]. The general formula for the generation is as follows:

$$SHA1(Key \bigoplus opad, SHA1(Key \bigoplus ipad, value)).$$

[6] whereas *opad* and *ipad* are predefined values [KCB97].

The random number generator uses a certain register in the TPM's non-volatile memory (RNG-state-register) to generate random numbers. Every operation that is performed by the TPM and requires random numbers receives them from the TPM's RNG.

The SHA-1 engine has to generate hash values. This algorithm has to be conform with [NIST95].

The power detection component keeps track of changes in the power-states of the overall system. This allows the TPM to restrict (or deny) certain commands to (in) specific states where the TP is physically constrained (e.g. Power-save-mode).

The non-volatile memory contains values that should be present within the TPM at all times — even after a re-boot. Examples for such values are: the storage root key (SRK), the private endorsement key and the TPME-identity key.

The volatile memory has to provide space for two authorisation sessions, two keys and various other values that are important for the current boot cycle.

[5] Considering a TPM that follows specification version 1.2.

[6] The symbol $\bigoplus$ equals an XOR operation.

Program Control Register The Program Control Registers (PCR) are used for storing integrity data about measurements. Since they are designed to store SHA-1 hash sums, they are of length 160 bits. The current system's state can be described as:

$$PCR_{(x,n+1)} = SHA1((PCR_{(x,n)})^\frown(SHA1(currentEvent))).$$

The new value (index $n+1$) of the PCR number x is generated by the hash sum of the concatenation of the SHA-1 hash sum of the event that was currently appended to the log file and the old PCR value (index n).

This allows us to carry a history with each register value, thus being potentially able to represent an unbounded number of log entries. At the beginning all PCRs are set to an initial power-up value. TCPA specification 1.1 requires 16 different PRCs ([TCPA03d] requires 24), where the first eight are used for base-functionality. We will discuss them more carefully later on.

Data Integrity Register The Data Integrity Register was within the non-volatile memory (TCPA main specification 1.1). It harbors a hash sum of a reference file. The reference values within this file can be used to enforce that the system is in a desired software state after a boot. In section 10 we will elaborate on the exact technique. In specification 1.2 the status of the old commands that could access the DIR are declared *deprecated*. Various changes were performed. Yet, a TPM that conforms to version 1.2 still has to support the old DIR commands. We will not elaborate further on these changes [TCPA03d].

7.2.6 Palladium or NGSCB

Palladium (Pd) is a codename for a set of security related features in the Windows environment. Recently it has been renamed to next-generation secure computing base (NGSCB) [Lin04]. This (industry-wide) standard has similar goals and approaches to the TCG. Thus, in the mass media these two approaches are often confused and mixed together in an illegitimate fashion. In this section we will describe architecture and features, and we will discuss the relation between NGSCB [Mic05, Mic03, Mic03b, Mic03a] on the one hand and TCPA version 1.1b [TCPA02, TCPA03e, TCPA03f, TCPA03b] and TCP version 1.2 [TCPA03d, TCPA03, TCPA03c, TCPA04, TCPA04b] on the other. Another name that occurs frequently when talking about trusted computing is *Intel's LaGrande Technology*[7]. During our examination of the hardware requirements of the NGSCB we will explain how *Intel's* project relates to NGSCB. Note that there is not yet a final draft of an NGSCB specification and that the material changes rapidly. So, for instance, the overall description of the security model

[7] ARM has a similar feature called *TrustZone*. Since Intel's approach is more prominent we will use their name to refer to the feature set they both represent [TZ04].

given in [Mic05, Mic03, Mic03b] seems to have changed according to [Bid04]. Since we were not able to obtain a proper technical description on the latest changes we describe the original architecture, which was valid until June 2004. However, the parts that may have changed will be highlighted.

The set of features mentioned earlier can be divided in four subcategories: curtained memory, signing data or code, storing data securely on an application level, and building a secure path from the I/O devices to NGSCB conforming applications.

1. The curtained memory enables the NGSCB application to wall off and hide data against other processes. Therefore, the processes can be certain that their data is not modified nor monitored.

2. Signing of data or code ensures that applications can certify or attest that they were created / produced in a trusted environment. By trusted environment is meant a locked-down NGSCB environment.

3. Storing data securely on an application level allows applications to mandate that its long-term stored data is only accessible by itself or by other trusted processes.

4. The secure path from the NGSCB conforming processes to its I/O devices should give the assurance that no other processes are modifying or overhearing the command stream.

These features should be used to provide the user with certain advantages. These include giving individuals and groups of users greater data security, personal privacy, system integrity, network security, and content protection (Digital Rights Management or DRM). These can be summarised in the following super classes:

- greater system integrity;
- superior personal privacy;
- enhanced data security.

Summarising the goal or mission of the NGSCB initiative: to help protect software against other software; that is, to provide a set of features and services that a software application can use to defend against malicious software, such as viruses or Trojan horses, running on the same machine. NGSCB does not aim to protect against elaborate hardware-based attacks.

Architecture of NGSCB

The following picture is taken from [CJPL02] and displays the main-structure:

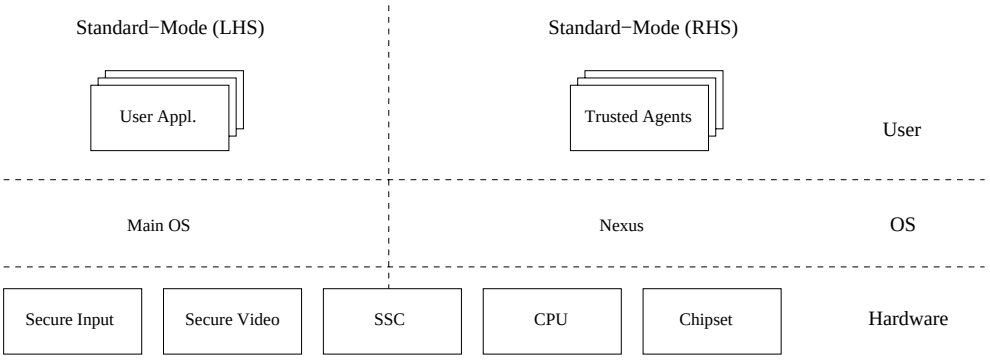

Figure 7.7: Architecture of the NGSCB

The Trusted platform according to Microsoft can be divided into two areas. One domain harbors all un-trusted operations and services (called the *Left-Hand-Side* or LHS in NGSCB terms) and the other includes the trusted processes (*Right-Hand-Side* or RHS). As shown in figure 7.7, the existing applications are running on the LHS and are not able to use the features of the security support component (SSC). This partition allows the design to cover various problems. One can always use non-trusted software in parallel to its trusted processes and it is possible to keep the size of the hardened operating system kernel (Nexus) to a minimum, since only very few operations that a normal operating system kernel should do are included. If a trusted process or the NEXUS requires the functionality of drivers or other operating system processes (such as file management), requests can be sent to the LHS.

The SSC represents the hardware component with the NGSCB interface that provides tamper-resistent storage for the secret keys and attestation functions. The minimum requirement that an SSC has to satisfy to meet the NGSCB criteria, include RSA public-key operations such as encryption, decryption, digital signature generation and verification, AES encryption/ decryption and SHA-1 hash computation. Additionally it hosts one RSA private and one AES symmetric key that are kept secret under any circumstances. These keys are unique and generated by the hardware-vendor. This hardware component is similar to the TPM in the TCPA version 1.1b specification. However, the TPM v1.1b does not include the required AES procedures. The TCG specification 1.2 remedies this deficiency. We could also say that the SSC is a TCG version 1.2 compliant device.

The Nexus or Trusted Operation Root (TOR) is the component that manages all trust-based functionality for their agents. More precisely, it is the mini-operating system kernel of the NGSCB isolated software stack. With the Nexus,

the NGSCB functionality is initialised during the booting process of the NGSCB hardware. The API of the Nexus enables NGSCB processes to establish process mechanisms for communicating with trusted agents and other applications, special trust services such as attestations of requests or sealing and unsealing of secrets. This module is verified before its start up. The precise procedure is similar to the secure boot description in the TCPA environment [TCPA02, Pea02]. First, the Nexus executable is loaded into the dedicated memory area, then a hash sum of the content of the memory block is generated. This hash is stored in a shielded location within the SSC (in TCPA terms called PCR). Finally, the execution handle is given to the Nexus.

The Trusted Agents (TA) can represent a program, a part of a program or a service that runs in the protected operating environment. Sometimes these processes are also called the Nexus Computing Agents (NCAs). These agents are using the Nexus as an interface to invoke security-related functions. Each TA is master of is own domain or sphere of trust. It is not required for them to trust other TAs in the same user space[8]. The combination of Nexus and TA provides following features: trusted data store, encryption services for applications, authenticated boot, facilities to enable hardware and software to authenticate themselves. Every TA is launched and managed by the Nexus. Before the TA can evolve into activity an integrity metric of its memory space is generated and stored in a secure register in the SSC. This process is identical with the verification procedure prior to executing the Nexus. This technique ensures the *chain-of-trust* as the TCPA calls this procedure.

Additional hardware is required to complete the approach. The demanded *strong process isolation* requires various changes to the standard Random Access Memory (RAM) management. The standard model divides the RAM in two areas: one is reserved for high priority processes such as the operating system (Ring-0) and the other can be utilized by user processes (Ring-3). It is normal practice that a user process (Ring-3) can call upon functions that reside within the Ring-0 memory space (*setuid*). Proper separation between these two memory spaces was (or is) only virtually possible. This condition enables attackers to gain full access over the complete platform. The NGSCB architecture circumvents this weak spot by introducing an additional bit that allows the system to switch between the trusted and un-trusted memory blocks. This leads to various changes in the common Central Processing Unit (CPU)[9].

[8] In [Bid04], the standard model seems to have changed slightly. Instead of one TA many TAs can share a common user space. This technique is called compartmentalisation. Unfortunately, at the time of writing no clear description of this revision was available.

[9] The following feature set is called, in Intel's terminology, *LaGrande Technology*.

- A mode flag has to be introduced to support the switch between the nexus mode and the standard mode.
- The possibility to decide between trusted and non-trusted memory pages has to be included. The trusted memory should only be addressable if the CPU has set its Nexus mode flag.
- CPU context switches have to be changed so that they can deal with the last two points.
- Various other smaller changes that are necessary to cope with the new operation mode [Mic03a].

There are various other changes to standard components, such as a modified *Chipset* and secure I/O devices. All I/O channels such as those from the keyboard to the computer or from the graphic engine to the monitor have to be encrypted. More specifically, all input devices are using TDES to ensure secrecy and each input signal is hashed into a cipher block chaining (CBC) message authentication code (MAC) which guaranties integrity [Mic03a].

At this point we will not elaborate further on the the NGSCB. The interested reader is referred to [Mic05].

7.2.7 Conclusion

In this chapter we have introduced two industry standards that aim to improve computer security on a large scale. The Trusted Computing Group (TCG) or TCPA, as it was formerly named, specifies a hardware component called the Trusted Platform Module (TPM) that should provide the rest of the platform with a root of trust. The TPM uses another element called the Core Root of Trust Measurement (CRTM) to ensure that not only the security relevant functions are working properly but also that executed code is what it pretends to be. The CRTM and its follower processes (called Measurement Agents) construct integrity metrics over important executed processes. Therefore, it is possible to determine the precise software state of the platform, and, furthermore, to communicate it to other (remote) entities. This allows an extension of the *chain-of-trust* beyond the boundaries of the local trusted platform.

The other approach, called next-generation secure computing base (NGSCB) takes this even further. It builds upon the TPM verion 1.2 and additional hardware to create a trusted environment. This trusted environment coexists with an area that harbours un-trusted processes or processes that are difficult to verify.

We also discussed the relations between the TCG and NGSCB approach and how Intel's LaGrande is related to them. NGSCB can be regarded as an extension of TCG's approach. We will not further discuss NGSCB explicitly. Nevertheless, by analysing the functionality of the TCPA's TPM we implicitly examine parts of

NGSCB as well. Elaborating on LaGrande Technology (or TrustZone) is beyond the scope of this thesis.

7.3 Introduction to Casper

As we have shown in the first half of this thesis, FDR proved to be very effective in spotting weaknesses in IDSs. Another big area that uses the FDR functionality is the verification of security protocols [LBH01].

In this domain one of the major issues is the definition of an appropriate CSP description of the protocol. The design process is time-consuming, requires profound skill with CSP and it is far from trivial not to introduce additional errors. Therefore, Lowe [LBH01] developed a compiler for the analysis of security protocols named *Casper*. *Casper* uses an easy–to–understand protocol description to generate a corresponding CSP model. The resulting CSP model can be used as FDR input. The protocol description language, also called *Casper* script, is more intuitive, especially to the protocol verification community, because of the similarities between the general description style used in the literature and the *Casper* script style.

To introduce *Casper* scripts, this subsection uses an example protocol taken from the standard distribution of *Casper* [LBH01]. First, we will explain the protocol and the corresponding *Casper* script. Then we will show how one can use FDR and *Casper* to obtain and interpret the results of the analysis.

The Yahalom protocol requires 5 transactions to establish a session key between two participants.

Message 1.	$Alice \rightarrow Bob$	:	$Nonce_A$
Message 2.	$Bob \rightarrow Server$	:	$\{Alice, Nonce_A, Nonce_B\}_{ServerKey(B)}$
Message 3a.	$Server \rightarrow Alice$	:	$Bob, \{Key_{AB}, Nonce_A, Nonce_B\}_{ServerKey(A)}$
Message 3b.	$Server \rightarrow Bob$	:	$\{Alice, Key_{AB}\}_{ServerKey(B)}$
Message 4.	$Alice \rightarrow Bob$	:	$\{Nonce_B\}_{Key_{AB}}$

In message 1 *Alice* sends *Bob* a nonce. In message 2 *Bob* transmits an encrypted message to the *Server*. This message contains the identity of the initiator (*Alice*) and a nonce from each participant ($Nonce_A$ and $Nonce_B$). *Bob* facilitates the $ServerKey(B)$ to ensure that no eavesdropper can obtain the content of this message. The *Server* generates a session key (Key_{AB}) for *Bob* and *Alice* and sends corresponding messages to *Alice* and *Bob*. Afterwards, *Alice* decrypts the message, extracts $Nonce_B$, encrypts $Nonce_B$ with the session key and sends this message to *Bob*.

7.3.1 Casper protocol description language

The script can be divided into various parts, as many source code files. The first part describes the protocol flow, the initial knowledge of the participants, the parameters and types used in the protocol, and the requirements that the protocol has to meet. The second part describes the system. It defines the names of the agents, message elements used, keys and functions involved. This definition will be used to generate the CSP model. This part also includes a description of the role that each participant will assume during a protocol run. Finally, it defines the initial knowledge and the abilities of the intruder.

Part one can be further divided into the section

- that declares the variables (called $\#Free\ Variables$).
- that describes the agents, their initial knowledge and their input parameters (called $\#Processes$).
- where the message handling and the overall information flow of the protocol is defined (called $\#Protocol\ description$).
- that specifies the requirements (called $\#Specification$).

We will briefly introduce them in order.

Free Variables The Free Variables section consists of the assignment of types to all variables that are used as well as the type definitions of the required functions. As we can see in our example, the variables a and b are of type $Agent$. Later on a and b will contain the name of the entities that execute the protocol, such as $Alice$ or Bob. Other types are $Nonce$ or $SessionKey$. It is important to state at this point that there does not exist a predefined typeset. More complicated type declarations are displayed in the last two lines. The type $ServerKey$ is in fact the result of the function $ServerKeys$. This function takes the name of an agent and returns the key that is shared (as a secret) between the participating server and the addressed agent.

```
#Free variables
a, b : Agent
s : Server
na, nb : Nonce
kab : SessionKey
ServerKey : Agent -> ServerKeys
InverseKeys = (kab, kab), (ServerKey, ServerKey)
```

The last line defines the encryption and decryption relationship between keys. For instance, in our example the key $ServerKey$ is inverse on itself; meaning one can use the key for encrypting and decrypting the message. As we will see later on, we have to design systems that use a public key infrastructure. Clearly the inverse key of the key used for the encryption in a public key system differs from the key that is used for decryption. First, we define the functions $PublicKey$, to generate the public keys, and $SecretKey$, to generate the secret keys. These functions receive the name of an agent and return the corresponding key. Finally, we define the inverse relationship between the keys PK and SK. The $Casper$ declaration for this would be:

```
PK : Agent  -> PublicKey \\
SK : Agent -> SecretKey \\
InverseKeys = (PK,SK)
```

Processes This part declares the roles of the participants, the knowledge that they already posses and the parameters that are provided externally. If we use the data-independence extension, this section defines the data that will be generated during runtime as well. The knowledge that is required for each role can be divided into the identification field, the variables that contain knowledge that is newly generated for each protocol run and the variables that contain data that remains unchanged. In our example we have the following:

```
#Processes
INITIATOR(a,na) knows ServerKey(a)
RESPONDER(b,s,nb) knows ServerKey(b)
SERVER(s,kab) knows ServerKey
```

The identifiers and values that are freshly generated in each protocol run are within the parenthesis. More specifically, in the above example, a assumes the role as the initiator of the protocol and uses the pre-generated nonce na in the current protocol run. The word $knows$ defines which knowledge of the initiator remains unchanged between the protocol runs, which in our case is the shared secret key between the trusted entity s and the initiator a.

Other roles can be the $RESPONDER$ or $SERVER$. More generally, the template of these declarations can be described as follows:

$$Rolename(ID, fv_1, ..., fv_x)\ \ knows\ \ nfv_1, ..., nfv_x$$

Protocol description This section describes the activities of the participants. It assumes the form of a message transmission and processing diagram. The description is straightforward:

```
#Protocol description
0.    -> a : b
[a != b]
1.  a -> b : na
2.  b -> s : {a, na, nb}{ServerKey(b)}
3a. s -> a : b,{ kab, na, nb}{ServerKey(a)}
3b. s -> b : {a, kab}{ServerKey(b)}
4.  a -> b : {nb}{kab}
```

Message 2, for instance, means that participant b sent participant s a message. This message consists of three encrypted elements (divided by a comma), the identity and two nonces (na and nb). This information block is encrypted by the result of the function $ServerKey(b)$. More generally, the information inside the first curly brackets contains the encrypted data and the key in the second curly brackets defines the encryption key. After this short example we give a small summary of the message description syntax:

- M_1, M_2 : message M_2 follows message M_1 without being tied together. Therefore, an intruder can split M_1 and M_2.
- $\{M\}\{K\}$: message M is encrypted with key K
- $h(M)$: message M is used as parameter to call function h. In many cases the function $h()$ will describe a hash function.
- $M_1(+)M_2$: message M_1 is combined with message M_2 by using a bitwise exclusive or operation. In other words, the $(+)$ operator can be used to model the Vernam encryption.
- $M\%x$: message M is transmitted to the receiver; yet, the receiver only stores the message inside variable x without prior inspection of the content. This is particularly useful whenever one agent functions as a relay for certain message parts.
- $x\%M$: message M is forwarded containing the value that has been assigned to variable x.

There is one other element in our script ($[a! = b]$). The Boolean expression inside the square brackets performs user defined verifications. In our case it checks whether agent a and b are different persons. If the Boolean expression returns false the protocol run will be suspended.

Specification The first part of the *Casper* script concluded with the specification of the security requirements that have to be met in order to be an error free protocol. Every specification that is stated in this section is converted by *Casper* into a corresponding CSP specification and related signalling events. There are six different types of specifications: *Secret*, *StrongSecret*, $WeakAgreement$, *Aliveness*, $NonInjectiveAgreement$ and *Agreement*. Since we only need two, we will omit the others. For more information, the interested reader is referred to [LBH01].

```
#Specification
Secret(a, kab, [b,s])
Secret(b, kab, [a,s])
Agreement(b, a, [na,nb])
Agreement(a, b, [kab])
```

- $Secret(ID, x, [A_1, , A_x])$ requires that, after each successful protocol run, the agent ID validly assumes that no other agent except the ones contained in the set that is defined by the square brackets knows the value within variable x. The drawback of this specification lies in the requirement that ID has to finish the protocol run.

- $Agreement(ID_1, ID_2, [x_1, , x_x])$ indicates that, if agent ID_2 thinks that he has finished a complete protocol run with agent ID_1, then agent ID_1 was previously using this protocol to communicate with ID_2. For the protocol run certain conditions have to hold. The first condition requires that both agents where aware of the role the counterpart assumed. The second requires that both agents were agreeing on the values bound to the variables x_1 to x_x. The last condition demands both agents to refer to the same protocol run.

The second part of the *Casper* script describes the actual system. It can be divided into four subcategories. The section that defines

- the real names, types of the agents and tokens involved (called $\#Actual\ Variables$).

- all required functions (called $\#Functions$).

- the roles of the actual participants (called $\#System$).

- the functionality and knowledge of the intruder (called $\#Intruder$).

Actual Variables This section defines the variables and the corresponding types that are used in the real system. The notation looks very similar to the $\#Free\ Variables$ part, except that the type description of functions is not included. In our example, for instance, we use the names $Alice$, Ivo and Bob as agent identities and Na and Nb as nonce. A binding convention for naming values does not exist, however it is preferred to use capital letters at the beginning to dinstinguish between abstract variables and real values.

```
#Actual variables
Alice, Bob, Ivo : Agent
Sam : Server
Kab : SessionKey
Na, Nb : Nonce
InverseKeys = (Kab, Kab)
```

Functions The function part specifies the types and functionality of functions. There are two types of functions: explicitly defined and symbolic defined functions. The explicit functions, as the name suggests, define every possible input and its corresponding outputs explicitly. Consider the following example:

$$TrueAgent(Owner) = true$$
$$TrueAgent(TPM) = true$$
$$TrueAgent(\_) = false$$

The function $TrueAgent$ accepts an identifier of an agent and verifies whether or not the name indeed belongs to a participating agent. The only two proper agent identifiers that are allowed in this certain protocol run are $Owner$ and TPM.

```
#Functions
symbolic ServerKey
```

The other type of function is symbolic, meaning that *Casper* generates the output and assigns a symbol to every output. In our example script the result of the function $ServerKey$ is defined in such a way.

System This part defines the roles of the participants and their parameters.

```
#System
INITIATOR(Alice, Na)
RESPONDER(Bob, Sam, Nb)
SERVER(Sam, Kab)
```

Intruder This section assigns the name of the agent that impersonates the intruder. Furthermore, it defines the values that are already known to the intruder, prior to the protocol run.

```
#Intruder Information
Intruder = Ivo
IntruderKnowledge = {Alice, Bob, Ivo, Sam, ServerKey(Ivo)}
```

7.3.2 Casper and FDR results

FDR uses the refinement checks that were generated by *Casper* to verify the protocol. FDR discovered more errors in the protocol. However, we will only describe one in detail. The following attack violated our first secrecy specification.

```
env.Alice.(Env0,Bob,<>)
send.Alice.Bob.(Msg1,Na,<>)
receive.Ivo.Sam.(Msg2,Encrypt.(ServerKey__.Ivo,<Alice,Na,Na>),<>)
send.Sam.Alice.(Msg3a,Sq.<Ivo,Encrypt.(ServerKey__.Alice,
                                                    <Kab,Na,Na>)>,<>)
receive.Sam.Alice.(Msg3a,Sq.<Bob,Encrypt.(ServerKey__.Alice,
                                                    <Kab,Na,Na>)>,<>)
send.Alice.Bob.(Msg4,Encrypt.(Kab,<Na>),<Na,Sam>)
send.Sam.Ivo.(Msg3b,Encrypt.(ServerKey__.Ivo,<Alice,Kab>),<>)
leak.Kab
```

Since FDR traces usually lack in readability, *Casper* provides us with the commands *interpret* and *linterpret*. The first command converts the FDR trace in a plain text description of the attack and the second command converts it into the message transmission style in Latex source (see figure 7.3.2).

Message 0. $\rightarrow Alice : Bob$
Message 1. $Alice \rightarrow I_{Bob} : Na$
Message 2. $I_{Ivo} \rightarrow Sam : \{Alice, Na, Na\}_{ServerKey(Ivo)}$
Message 3a. $Sam \rightarrow I_{Alice} : Ivo, \{Kai, Na, Na\}_{ServerKey(Alice)}$
Message 3a'. $I_{Sam} \rightarrow Alice : Bob, \{Kai, Na, Na\}_{ServerKey(Alice)}$
Message 4. $Alice \rightarrow I_{Bob} : \{Na\}_{Kai}$
Message 3b. $Sam \rightarrow I_{Ivo} : \{Alice, Kai\}_{ServerKey(Ivo)}$
The intruder knows Kab

Message 0 states that *Alice* and *Bob* want to run a session. In message 1 *Alice* sends the first nonce to *Bob*. However, the intruder intercepts this message and uses the nonce to send a session key generation request to the server. The session key generation request (message 2) consists of the identity of *Alice* and

the intercepted nonce Na. This message is encrypted with the $ServerKey(Ivo)$. The server generates a session key (Kai) and sends notification messages to *Alice* (message 3a) and to the intruder (message 3b). The message that is targeted at *Alice* is intercepted by the intruder. In message 3a' the intruder pretends to be the server and relays a modified version of message 3a to *Alice*. The modification ensures that *Alice* does not recognise that the key Kai was generated to establish a session between *Alice* and Ivo. Afterwards, *Alice* sends an acknowledgement to *Bob* (message 4). This acknowledgement is intercepted by the intruder. In the last message the server sends the intruder the identity of the protocol initiator and a session key encrypted with $ServerKey(Ivo)$.

Chapter 8

Authorisation protocols

The TCPA enforces various authorisation protocols that can be used to securely create protected objects. Every protected object possesses an authorisation secret. The five standard protocols that are required to fulfil the TCPA specification not only deal with the creation of protected objects but also the establishment of secure channels to the TPM. These channels can be used for transmitting a proof of knowledge or to change authorisation data of an object in a secure manner.

The TCPA puts special attention on the modularisation of these protocols as well as their re-usability; for instance the Object Independent Authorization Protocol (OI-AP) and the Object Specific Authorization Protocol (OS-AP) work as a first building block for more advanced protocols such as the Authorisation Data Change Protocol (ADCP). [TCPA02] states that these building blocks can be used to generate more elaborate protocols. In Chapter 11 we pick up this claim and use them to generate a digital rights management protocol.

In this chapter we will describe the five basic authorisation protocols:

The Object Specific Authorization Protocol (OS-AP) uses one value of authentication data (the so called secret) for many authentications on the same target object without requiring a re-authentication.

The Object Independent Authorisation Protocol (OI-AP) uses multiple authorisation secrets to access many target objects in the same session.

The Authorization Data Insertion Protocol (ADIP) binds new authorisation data during the creation of a particular object to this object; during creation the user must show knowledge of the so called parent authorisation secret, hence the user must establish an OS-AP session.

The Authorization Data Change Protocol (ADCP) is used to change the secret that is used for authorisation to access a specific object; after applying this command the object's secret is changed, however the owner of the TPM can still access this object.

The Asymmetric Authorization Change Protocol (AACP) is used to prevent the owner of the TPM having access to the object; this is of significance whenever we store data on a trusted platform and we don't want to reveal the content to the owner of the platform (e.g. Digital Rights Management).

We will look from three different (levels) perspectives at these protocols: first, the message passing level, second the operational level (the command level) and finally the composability level that shows whether the various building blocks can be combined without violating the security.

The message passing level will look at the protocols in the classical way. It will verify whether the claims in the specification are justified.

The command level will not only investigate the succession of TPM operations involved in triggering and processing the protocol messages, but also the part of the TPM that are necessary for these commands (see section 9).

The last part of the investigation deals with the composability of the protocols. More precisely we will justify whether it is safe to use OS-AP and OI-AP as building blocks for ADCP. We hope to derive more general approaches form this specific result.

Note in order to conduct a proper analysis (due to the state space problem) of the protocols we prune away certain features. We then prove that these abstractions did not interfere in a negative way with the evaluation of our protocol. In this thesis we will not describe each proof for every protocol, otherwise we would describe very similar proofs again and again. Instead we will describe all necessary protocols as the TCPA defines them and choose one amongst them to exercise a full analysis; our protocol of choice is the Object Specific Authorisation Protocol (OS-AP).

The chapter will close with a discussion about potential protocols that can be built upon these basic building blocks.

8.1 Object Specific Authorisation Protocol

The Object Specific Authorisation Protocol (OS-AP) establishes an authentication session for one protected object. It uses only one authorisation value for many commands that operate on the same object without requiring a *re-authentication*. It does so by establishing a session key or so called ephemeral secret. This protocol is efficient in environments where a stream of commands requires access to a single protected object. Once the ephemeral secret is established all commands use the ephermal secret instead of the real authentication value to gain access. The protocol itself can be divided in three parts:

- the establishment and generation of the session key;

- the submission of the command and its feedback;
- the continuing handshake of command and return parameter submissions — if the authenticated session should continue.

The last element bears the difficult part of the protocol verification. FDR has to explore all states of a protocol that has (potentially) an infinite supply of commands and different nonces. A supply of infinitely different nonces renders the complete system into a process with infinitely different states.

8.1.1 Description

In this section we will describe the OS-AP protocol in its initial version, without abstractions.

The first two messages are concerned with the establishment of the shared secret. Messages 3 submits the command and message 4 closes this cycle by transferring the response of the command back to the initiator. The last part, messages 5 and 6, are only available if the $continueAuthSession$ tags (in message 3 and 4) are set to true; indicating that there are more commands to process.

Message 1.	$Owner \rightarrow TPM$	:	$TPM\_OSAP, prentHandle, nonceOddOSAP$
Message 2.	$TPM \rightarrow Owner$	:	$authHandle, authLastNonceEven$
			$nonceEvenOSAP$
Message 3.	$Owner \rightarrow TPM$	:	$tag, paramSize, ordinal, inArgOne, inArgTwo$
			$authHandle, nonceOdd, continueAuthSession$
			$HMAC(sharedSecret, D_1)$ $where$
			$D_1 = \langle InArgHashOne, InArgHashTwo \rangle$
Message 4.	$TPM \rightarrow Owner$	:	$tag, paramSize, returnCode, outArgOne$
			$nonceEven, continueAuthSession$
			$HMAC(sharedSecret, D_2)$ $where$
			$D_2 = \langle OutArgHashOne, OutArgHashTwo \rangle$

Message one passes along the TPM_OSAP command with the $parentHandle$, that identifies a encryption key and the nonce $nonceOddOSAP$. On reception of this message the TPM creates and allocates the necessary resources for the expected authentication session. It generates two nonces called $AuthLastnonceEven$ and $nonceEvenOSAP$, it links the session resources to $authHandle$ and calculates the session key by building an HMAC of the authorisation secret of the target object and the values $nonceEvenOSAP$ and $nonceOddOSAP$. Then, in message 2, the TPM returns the label of the session called $authHandle$ with the two newly generated nonces. Thereafter the initiator saves the received information, calculates the session key, generates nonce $nonceOdd$ and computes the parameter $inAuth$ by calculating a HMAC of the shared secret and

all input parameters ($InArgHashOne$) with the authorisation setup parameters ($InArgHashOneTwo$). In message 3 the initiator submits the command and various other required parameters. The TPM verifies the authorisation session ID ($authHandle$) and recalculates the value of $inAuth$ and compares it with the recently received value. Finally the command is executed, the return code is generated and the value $resAuth$ is produced by the application of the HMAC function onto the shared secret, the outgoing parameters ($OutArgHashOne$) and the outgoing setup parameters ($OutArgHashTwo$). After receiving message 4 the initiator recalculates the value of $resAuth$ and compares this value with the recently received data, thus verifying the return code and the output parameters.

In message 3 the owner can decide whether he wants to execute another command upon the same object. If he chooses so, the value *continueAuthsession* has to be $TRUE$. In message 4 the TPM has the option to deny the request for continuation. We will see such a rejection in the ADIP and in the ADCP. If the TPM chooses to grant the request the following two messages will be submitted.

Message 5. $Owner \rightarrow TPM$: $tag, paramSize, ordinal, inArgOne$
$inArgTwo, nonceOdd, continueAuthSession$
$HMAC(sharedSecret, D_3)$ *where*
$D_3 = \langle InArgHashOne, InArgHashTwo \rangle$

Message 6. $TPM \rightarrow Owner$: $tag, paramSize, returnCode, outArgOne$
$nonceEven, continueAuthSession$
$HMAC(sharedSecret, D_4)$ *where*
$D_4 = \langle OutArgHashOne, OutArgHashTwo \rangle$

Where :
$sharedsecret = HMAC(key.usageAuth, nonceEvenOSAP, nonceOddOSAP)$
$InArgHashOne = SHA1(ordinal, inArgOne, inArgTwo)$
$InArgHashTwo = authLastNonceEven, nonceOdd, continueAuthSession$
$OutArgHashOne = SHA1(returnCode, ordinal, outArgOne)$
$OutArgHashTwo = nonceEven, nonceOdd, continueAuthSession.$

The initiator generates a nonce ($nonceOdd$) and computes the HMAC of the shared secret, the input parameters ($InArgHashOne$) and the authorisation setup parameters ($InArgHashTwo$). In message 5 the initiator submits the command and all relevant parameters to the TPM. The TPM performs the same actions as after receiving message 3, except that it does not verify the authentication handle. Finally the return code and additional parameters are returned to the initiator (message 6). Only if *both* parties want to continue this session the computation starts again with the calculations that are required to produce message 5. This allows the TPM to stop the session even if the user wants to continue. The nonce $nonceEven$ remains in the system to link message 7 to message 6.

8.1.2 Basic model

In this section we will describe the procedure to perform a complete analysis of the OS-AP protocol. The analysis can be divided in four steps:

1. pruning away unnecessary data;
2. proving that these abstractions are sound;
3. building an infinite state model of the protocol;
4. modifying the CSP model so that we can use FDR for the verification.

The reduced OS-AP In this section we described the version that is specified by the TCPA. From this we will derive a reduced protocol that can be used for our analysis. We will look at every message and explain why parts are redundant and how one can prove that they are redundant for the security of the overall protocol. All our abstractions will be fault-preserving. Note a fault-preserving transformation does not prune away any weaknesses of the original protocol; thus if the simplified protocol refines a particular security specification then the initial version does also.

$$\begin{array}{lrcl} \text{Message 1.} & Owner \rightarrow TPM & : & TPM\_OSAP, parentHandle, nonceOddOSAP \\ \text{Message 2.} & TPM \rightarrow Owner & : & authHandle, authLastNonceEven \\ & & & nonceEvenOSAP \end{array}$$

In the first two messages we omitted the tag TPM_OSAP and the value $nonceEvenOSAP$. The tag TPM_OSAP only indicates that the owner wants to launch an OS-AP session. This may be important if we simulate more protocols in one model, however since this analysis only focuses on one protocol the TPM does not require this signal value. Furthermore TPM_OSAP does not occur on any other message and it does not influence the protocol flow in any way. Additionally this field is of a distinctive type, hence can not be used to fake the content of other data fields. The function that removes the value TPM_OSAP from the protocol satisfies the 3 conditions that are required in order to be a fault-preserving transformation. It is plain to see why conditions 3 and 2 are satisfied by a function that erases the fact TPM_OSAP from every message. This value never occurs in IIK or IIK' as well as it is never an element of the $AgreementSet$. Condition 1 is met since TPM_OSAP is not bound to other protocol messages (no fact needs the value to draw conclusions upon other necessary values).

The second abstraction is more vital since this nonce will be used to ensure that the session secret cannot be replicated. We have to be certain that the freshness property is not violated by our abstraction. Looking at the third message of the protocol we see four nonces. Two are included within the shared secret.

They link every chain of commands (the complete run from message 3 onwards) to a single protocol run. The other two nonces are included in the SHA1 hash sum. These two nonces ensure a proper succession of commands. Hence it is impossible to reply or swap a command within a single protocol run. The following abstracted protocol description gives an overview of this scenario:

$$\begin{array}{lll}
\text{Message 1.} & A \rightarrow B & : N_1 \\
\text{Message 2.} & B \rightarrow A & : N_2, N_3 \\
\text{Message 3.} & A \rightarrow B & : N_4, SHA1(HMAC(sharedSec, D_1), N_3, N_4) \\
 & & \quad where\ D_1 = \langle N_1, N_2 \rangle \\
\text{Message 4.} & B \rightarrow A & : N_5, SHA1(HMAC(sharedSec, D_1), N_5, N_4) \\
 & & \quad where\ D_1 = \langle N_1, N_2 \rangle
\end{array}$$

We have to show that after our abstraction each message (from 3 onwards) is still linked to a particular protocol run (to message 1 and 2) and that an intruder can not alter the order of the commands. By eliminating the value N_2 the link between the messages that follow message 3 are not anymore directly linked to message 2. However nonce N_3 takes over that purpose. It links message three to two. In message 4 N_4 links message 3 and 4 together. The responder exchanges the N_3 with a new value. Thereafter N_5 ties message 4 to 5 and so on. The paper [BL02] examines a similar situation.

Note nonces fulfil various other purposes therefore the statement given above only represents an outline why it is sensible to abstract this field away and why it is highly likely that it does not introduce new attacks. For a thorough analysis our explanation is not sufficient. The only thing we really have to guarantee is that this transformation does not lose attacks. Lowe and Hui already showed that such an abstraction is fault-preserving since it abides by the three necessary conditions.

$$\begin{array}{lll}
\text{Message 3.} & Owner \rightarrow TPM & : tag, paramSize, ordinal, inArgOne, inArgTwo \\
 & & \quad authHandle, nonceOdd, continueAuthSession \\
 & & \quad HMAC(sharedSecret, D_1)\ where \\
 & & \quad D_1 = \langle InArgHashOne, InArgHashTwo \rangle \\
\text{Message 4.} & TPM \rightarrow Owner & : tag, paramSize, returnCode, outArgOne \\
 & & \quad nonceEven, continueAuthSession \\
 & & \quad HMAC(sharedSecret, D_2)\ where \\
 & & \quad D_2 = \langle OutArgHashOne, OutArgHashTwo \rangle
\end{array}$$

Message 3a, 3b and 3c communicate various fields that can be neglected or combined; in particular the values $ordinal$, $inArgOne$ and $inArgTwo$. At this point of our analysis we do not consider internal transactions within the client or the TPM. Therefore we do not specify which command the owner issues to access the protected object. If we can show that the intruder is unable to intercept the messages 3a, 3b, 3c and to exchange the values ($ordinal$, $inArgOne$ and $inArgTwo$)

with his own faked values, we may very well drop these fields. Assuming that the attacker is successful with such an attack, he can, undetected by the TPM, modify one of the important values. The TPM, before it executes the command, verifies the input of message 3 by recalculating the transmitted HMAC (message 3c). The HMAC contains the *sharedsecret*, which is only known by the TPM and the legitimate owner, and all input parameters except *tag* and *paramSize*. If we assume that the hash is generated by a collision free function ([NIST95]) and we know already that the key used for building the HMAC is only known by the owner and the TPM, it is easy to see that the intruder can not recalculate the hash so that it would fit the new message. Therefore under any circumstance the HMAC will (nearly) never match to a modified message. The above mentioned three conditions hold for this transformation (see TPM_OSAP example). The formal proof can be found in [HL01].

In order to provide a better foundation for later verifications we decided to combine the fields *ordinal*, $inArgOne$ and $inArgTwo$ in message 3 to one value. According to Lowe and Hui this is a fault-preserving transformation. We apply the same coalescing action to $returnCode$, *ordinal* and $outArgOne$. Afterwards we use another fault-preserving function that renames the result of the mergers to *clientinput* (for *ordinal*, $inArgOne$ and $inArgTwo$) and *clientoutput* (for $returnCode$, *ordinal* and $outArgOne$).

The other parameters that can be neglected are the *tag* and the *paramSize* field. The *tag* value because the owner and the TPM do not exchange valuable information in this field (this may be different if we use another angle to look at the protocol, see chapter 9). Additionally, both participants can distinguish between the types of the data elements that were transmitted. Hence it is not possible for the intruder to use a value, transmitted by the *tag* field, as a value in another field. The same reasoning holds for the *paramSize* field. Again, if we would look at the command level this parameter may very well not be negligible. The formal justifications are similar to the one given in the TPM_OSAP case. Note, to further reduce the state space, we also removed the internal hash of *sharedsecret*. For a formal justification see [HL01]. This leads to following CSP definition.

The initiator The CSP description of the *Owner* process consists of various other sub-processes. We will describe them and a general template for the infinite series of processes that allow the system to handle a continuous stream of commands. The basic *Owner* process needs access to an infinite sequence of nonces and commands. Instead of initialising the process with such sequences we establish events that supply the processes with as many nonces and commands as they require. This design eases the modifications that are necessary to reach a model with a finite amount of states. The process that synchronises on the event $pickNonce$ and keeps track of the nonces that were already in use is called

the $NonceManager$. The process $commandInputManager$ has the same purpose only that it manages the unbounded command line submission. We will describe these processes later. The procedure of sending and receiving the first four messages, on the initiator's side, is performed by following process:

$$
\begin{aligned}
&Owner_0(O, T) = \\
&\quad pickNonce?na_1 \rightarrow \\
&\quad send\,.\,T\,.\,O\,.\,(Msg_1, \langle na_1 \rangle) \rightarrow \\
&\quad \Box\, authhandle : AuthorisationHandle,\, nb_1 : Nonce \bullet \\
&\quad\quad receive\,.\,O\,.\,T\,.\,(Msg_2, \langle authhandle, nb_1 \rangle) \rightarrow \\
&\quad\quad pickNonce?na_2 \rightarrow pickInput?clientdatainput_1 \rightarrow \\
&\quad\quad send\,.\,T\,.\,O\,.\,(Msg_{3a}, \langle clientdatainput_1, authhandle, na_2 \rangle) \rightarrow \\
&\quad\quad send\,.\,T\,.\,O\,.\,(Msg_{3b}, \langle SHA1(SK(O,T), na_1, clientdatainput_1, \\
&\quad\quad\quad\quad nb_1, na_2) \rangle) \rightarrow \\
&\quad \Box\, clientdataoutput_1 : TOutput,\, nb_2 : Nonce \bullet \\
&\quad\quad receive\,.\,O\,.\,T\,.\,(Msg_{4a}, \langle clientdataoutput_1, nb_2 \rangle) \rightarrow \\
&\quad\quad receive\,.\,O\,.\,T\,.\,(Msg_{4b}, \langle SHA1(SK(O,T), na_1, \\
&\quad\quad\quad\quad clientdataoutput_1, nb_2, na_2 \rangle) \rightarrow \\
&\quad\quad forget\,.\,O\,.\,na_2 \rightarrow \\
&\quad\quad forgetInput\,.\,O\,.\,clientdatainput_1 \rightarrow \\
&\quad\quad Owner_1(O, T, na_1, nb_2).
\end{aligned}
$$

We decided to put as many messages as possible in the initial owner sub-process $Owner_0$. This process is initialised by the its own and the TPMs identity. The owner picks a nonce via $pickNonce$ and uses that nonce to transmit the first message. Afterwards the owner waits to receive the answer of the TPM. Once the content of message 2 has been received $Owner_0$ requests another nonce and its first command ($clientdatainput_1$). After sending and receiving the next messages we raise the forget events. These events are used to notify the $NonceManager$, the $commandInputManager$ and the $commandOutputManager$ that an element of their domain has expired (e.g. nb_2 for the $NonceManager$). We will describe the exact mechanism later. Finally we parameterise $Owner_1$ with the necessary information to continue the protocol run.

$$
\begin{aligned}
&Owner_1(O,\, T,\, na_1,\, nb_2) = \\
&\quad pickNonce?na_3 \rightarrow pickInput?clientdatainput_2 \rightarrow \\
&\quad send\,.\,T\,.\,O\,.\,(Msg_{5a},\, \langle clientdatainput_2, na_3\rangle) \rightarrow \\
&\quad send\,.\,T\,.\,O\,.\,(Msg_{5b},\, \langle SHA1(SK(O,T), na_1,\, clientdatainput_2, \\
&\qquad\qquad nb_2,\, na_3)\rangle) \rightarrow \\
&\quad \Box\, clientdataoutput_2 : TOutput,\, nb_3 : Nonce \bullet \\
&\quad\quad receive\,.\,O\,.\,T\,.\,(Msg_{6a},\, \langle clientdataoutput_2, nb_3\rangle) \rightarrow \\
&\quad\quad receive\,.\,O\,.\,T\,.\,(Msg_{6b},\, \langle SHA1(SK(O,T), na_1,\, clientdatainput_2, \\
&\qquad\qquad nb_3,\, na_3)\rangle) \rightarrow \\
&\quad\quad forget\,.\,O\,.\,na_3 \rightarrow \\
&\quad\quad forgetInput\,.\,O\,.\,clientdatainput_2 \rightarrow \\
&\quad\quad Owner_1(O,\, T,\, na_1,\, nb_3).
\end{aligned}
$$

$Owner_1$ performs another handshake between the TPM and the $Owner$. The process obtains the next nonce (na_3) and the second command ($clientdatainput_2$) and sends the command request in message 5a and 5b. After receiving (message 6a and 6b) the feedback of the TPM, the initiator forgets the nonce na_3 and the command $clientdatainput_2$. Finally, it loops back to submit the next command.

The responder Since the process Tpm_0 is the precise counter process of $Owner_0$ we will omit a description of the CSP code.

The nonce manager The process $NonceManager$ is parameterised by an infinitely long sequence of nonces. Whenever one of the processes chooses to pick a nonce, the head of the sequence is communicated back to the caller. To circumvent repetitions, the tail of the initial sequence is used to call the next nonce manager.

$$NonceManager(xs) = pickNonce\,.\,head(xs) \rightarrow NonceManager(tail(xs)).$$

The processes $commandInputManager$ and $commandOutputManager$ have the same structure. Note, for our final model we have to modify the nonce manager in such a way that only a limited amount of nonces is sufficient (nonce recycling).

The intruder The intruder utilises the Dolev-Yao [DY83] approach. The resulting process is capable of monitoring all communicated OS-AP messages, dropping messages during transit, pretending to be a certain participant in the network and to inject self-forged messages at will. These messages however have to be built upon prior knowledge. This knowledge can stem from collected messages or form deductions that were performed upon collected information. These $Deductions$ represent the core of the intruder. The set $Deductions$ consists of

tuples e.g. (x, Y), whereas Y indicates the information that is necessary for the intruder to retrieve information x. For instance if we consider the creation of a hash sum the Y consist of the hash function and the value that has to be hashed. Following this example, x, would then be the result of the result of the hash calculation. The model of the intruder:

$$
\begin{aligned}
&Intruder(IK) = \\
&\quad receive?O\,.\,T\,.\,x \rightarrow Intruder(IK \cup \{x\}) \\
&\quad \sqcap \\
&\quad \sqcap x : IK, O, T : Agent \bullet send\,.\,O\,.\,T\,.\,x \rightarrow Intruder(IK) \\
&\quad \sqcap \\
&\quad \sqcap (x, Y) : Deductions,\, Y \subseteq IK \bullet infer\,.\,(x, Y) \rightarrow Intruder(IK \cup \{x\}).
\end{aligned}
$$

The *Intruder* process is initialised by the initial knowledge (IK). The set IK contains the identifier of the other participants ($Owner$ and TPM), enough data to launch an OS-AP session (e.g. nonce), the identifier of the protected object that belongs to the intruder and its corresponding authorisation secret. The event *receive* is used to monitor the conversations between the owner and the TPM. After performing the *receive* operation the intruder adds the collected message x to its knowledge. The process *intruder* can also send, via the *send* event, every fact that is within its knowledge base. Finally, to gain new information, the intruder can perform deductions upon its collected knowledge. This $infer$ operation is only available if the set Y is a subset or equal to its knowledge base.

Note that this design only shows the semantic model of the intruder. The CSP implementation looks different. Roscoe and Goldsmith [RSG+01] designed a highly efficient version of the intruder. In their model every element that possibly can be learned by the intruder is represented by its own two state process. One state represents that the intruder does not know the fact; the other that he knows. For further explanation of this model see [RSG+01].

The system The processes are connected with other relevant processes according to Figure 8.1.

The *Initiator* and *Responder* sending to and receiving messages from the *Intruder*. The *NonceManager* is connected via the *pickNonce* event to the *Owner* and the TPM. Similarly *commandInputManager* and *commanOutputManager* are connected via the events *pickInput* and *pickOutput* to their target processes.

The specification The TCPA designed the protocol to achieve proper authorisation between the TPM and the client. This goal can be further divided: first providing secrecy for the transmission of the authorisation secret and second ensuring that both participants are properly authenticated to each other.

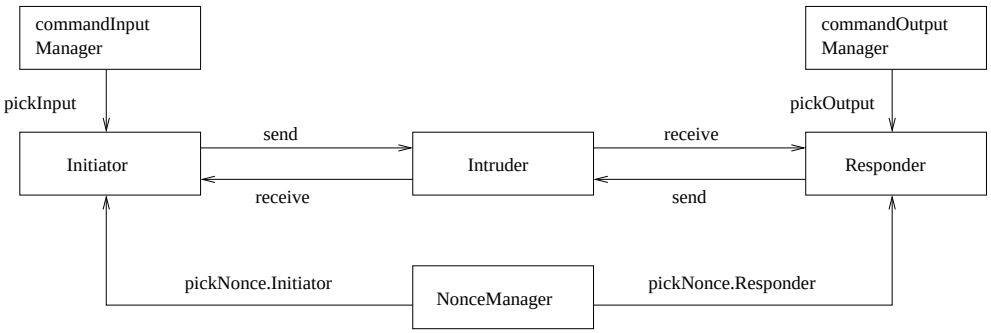

Figure 8.1: OS-AP communication flow

The first part considers whether the intruder can learn the authorisation secret that the TPM and the client share. After we complete our system according to our network layout (see figure 8.1) we hide every event except the leak event of the intruder and a signal event that indicates what agents are performing the protocol run ($signal.Claim\_Secret.Owner.sharedsecret.TPM$). The intruder can only engage in a *leak* event if the fact, he is about to reveal, is within a set called $ALL\_SECRETS\_DI$. This set contains, after renaming, all elements that are part of the secrecy specification. Thus the specification allows a leak event only if the intruder participates actively in the protocol run, e.g. as a legitimate agent. More formally we can say:

$$\forall tr : Traces \bullet signal.Claim\_Secret.Owner.sharedsecret.TPM \text{ in } tr \Rightarrow tr \downarrow leak.sharedsecret = \langle\rangle$$

If we can show that $traces(System_{secret}) \subseteq traces(Specification_{secret})$ we would have proven that OS-AP holds its claim regarding secrecy.

The remaining specification considers whether the commands send to the TPM are properly authorised. If we use the same approach as before we can use following trace specification to enforce such a behavior:

$$\begin{aligned}&\forall tr : Traces\,; na_1, na_2, nb_1 : Nonces\,; cdatain_1 : CInput;\\ &\quad cdataout : TOutput\,; SK(O,T) : SecretKeys;\\ &\quad O = Owner\,; T = TPM \bullet\\ &\quad tr \downarrow receive.O.T.\langle SHA1(SK(O,T), na_1,\, cdataout_1,\, nb_2,\, na_2)\rangle \leq\\ &\quad tr \downarrow send.T.O.\langle SHA1(SK(O,T), na_1,\, cdatain_1,\, nb_1,\, na_2)\rangle\end{aligned}$$

This specification centers around message 3b and 4b in our OS-AP description. It expresses that whenever the TPM returns the output of a valid command execution (message 4b) then there was a valid request to execute such a command. The CSP version of this specification consists only of a sequential process that

allows to send message 3b and only then permits to engage in the submission of message 4b. As before all irrelevant behaviors are hidden during the refinement check.

8.1.3 The final model

So far we have reduced the structure of the protocol by applying fault-preserving transformations. This leaves us with one problem — the model still has an unbounded state space. This problem can be divided in two subproblems: first the nonces and second the commands. As mentioned in the IDS part of this thesis, Roscoe and Lazic [Ros98] invented a method to reduce the scope of certain types without losing relevant detail. We used this method to design a model that has an infinite amount of new nonces, command inputs and outputs, whereas the types themselves remain finite. To restrict the supply of commands we use Theorem 2; thus two distinct commands are sufficient. The problem with the supply of nonces is different, since the nonces have to be different. Therefore we will use a technique presented in [Bro01], that allows us to establish a recycling mechanism that re-uses nonces once they are no longer in use by the TPM and by the owner. Clearly every nonce will still be in use by the intruder, since every message is stored in the intruder's knowledge base. In certain situations this can cause serious problems. Hence, we establish a nonce manager process that is capable of converting nonces within the intruder's knowledge. Once a nonce is no longer required the corresponding nonce in the intruder's knowledge is projected to a predefined value. The transformation process has to be performed for every fact that contains that particular nonce. The modifications on our CSP model were minor, since we had already included the $forget$ and $pickNonce$ events. [RB99, Bro01] show that this technique is sound.

The last change that converts our initial description in a finite system is the reduction from the infinite amount of sender and receiver processes. The structure of messages 5 and 6 does not differ of the structure of messages 7 and 8 (and so forth). Hence after reaching the final event of process $Owner_1$ we can simply loop back to itself. We only have to be careful about the remaining nonces. The same applies to the TPM_1 process.

8.1.4 Results

After applying all abstractions we used FDR to verify whether the claims of the TCPA where true. Our analysis revealed following problem:

Message $\alpha.0.$	$\rightarrow Alice$	:	TPM
Message $\alpha.1.$	$Alice \rightarrow I_{TPM}$	:	$Na2$
Message $\beta.1.$	$I_{Bob} \rightarrow TPM$	:	$Na2$
Message $\beta.2.$	$TPM \rightarrow I_{Bob}$	:	$Authhandle, Nb$
Message $\alpha.2.$	$I_{TPM} \rightarrow Alice$	:	$Authhandle, Nb$
Message $\alpha.3.$	$Alice \rightarrow I_{TPM}$	:	$Clientdatainput, Authhandle, Na$
Message $\alpha.3a.$	$Alice \rightarrow I_{TPM}$	:	$f(Sk, Na2, Clientdatainput, Nb, Na)$
Message $\beta.3.$	$I_{Bob} \rightarrow TPM$	:	$Clientdatainput, Authhandle, Na$
Message $\beta.3a.$	$I_{Bob} \rightarrow TPM$	:	$f(Sk, Na2, Clientdatainput, Nb, Na)$
Message $\beta.4.$	$TPM \rightarrow I_{Bob}$	:	$Clientdataoutput, Nb3$
Message $\beta.4a.$	$TPM \rightarrow I_{Bob}$	:	$f(Sk, Na2, Clientdataoutput, Nb3, Na)$
Message $\alpha.4.$	$I_{TPM} \rightarrow Alice$	:	$Clientdataoutput, Nb3$
Message $\alpha.4a.$	$I_{TPM} \rightarrow Alice$	:	$f(Sk, Na2, Clientdataoutput, Nb3, Na)$

The attack can be divided in two protocol runs (α and β). First *Alice* decides to communicate with the TPM and sends the necessary nonce $Na2$. The intruder impersonates the TPM and forwards (as *Bob*) the intercepted $Na2$ to the proper TPM. The real TPM responds appropriately in message $\beta.2$. This information is forwarded to *Alice* (message $\alpha.2$) and she responds by sending the command message ($\alpha.3$ and $\alpha.3a$). This command request is used to launch the same command on the real TPM (message $\beta.3$ and $\beta.3a$). In the last stage of the protocol the real TPM returns the output (message $\beta.4$ and $\beta.4a$) of the command and the intruder uses these messages to impersonate the TPM in the messages $\alpha.4$ and $\alpha.4a$. At this point the TPM believes it was running the protocol with *Bob*, whereas it took the role of the responder. On the other side, *Alice* believes she has successfully submitted one command to the TPM.

There are other attacks, they are all based on the same problem. The receiver can not determine whether he was the designated target or not, nor is the sender able to include information about the origin of the message. At this point we omit the description of the reverse attack. To fix the protocol the identities of both participants have to be included in message 3 and 4. To prevent the intruder from changing the IDs, they should be included in the $SHA1$ hash sum. It is worth noting that we do not need to include the identities in subsequent command submission.

8.1.5 Discussion

In this chapter we used a technique presented in [BL02] to verify the OS-AP. First we pruned away all irrelevant fields of the initial protocol. Second we showed why

these abstractions seem to be sensible and not introduce false positives. However far more important for such an analysis was to show that our transformations were fault-preserving. We used the theorems and pre-made proofs of Hui and Lowe [HL01] to show that our reduced protocol still contained all security violations. Third we designed a *Casper* script according to our reduced OS-AP. Fourth we modified the CSP model so that it was able to cover the continuous command stream. Finally we showed how one can project a infinite state space model on to a finite one.

Our analysis revealed various attacks, of which we picked one and discussed it in detail. All attacks stemmed from the same problem. The messages that carry the command call to the TPM (responder) do not indicate what the desired destination is nor do they reveal who the sender is. This problem exists as well with all messages that communicate the results of a successfully executed command.

One may argue that this may be only a minor inconvenience, especially when considering the basic authorisation protocols that are build upon OS-AP. The attacker is not able to change the messages he can only re-direct them. So if the high-level protocol that is using the fields *clientdatainput* and *clientdataoutput* encrypts its sensitive content, no serious damage can occur. In the worst case the initiator thinks that the command was not executed, whereas in fact the TPM successfully processed the request. This attack, in its ultimate result, would not differ from the ubiquitous situation that the message that includes the response of the TPM can be lost between the two participants.

A slightly different situation arises if we look at the command *TPM_Unseal*. This command decrypts the object and transmits in plain text the content back to the caller. The specification does not specify the context in which the operation has to be executed (e.g. over encrypted connection or not). Hence there may be implementations where this can cause problems.

We will see in the sections about the ADIP, ADCP and AACP that this is exactly the case.

On the other side the TCPA main specification [TCPA02] explicitly states that

> The OS-AP allows establishment of an authentication session for a single entity.

If protocol designers take this claim for granted it may very well be that they do not include elements that, amongst other things, subsequently lift the resulting protocol into the class of authentication protocols. We will further discuss the seriousness of this attack in our DRM section. For the rest of the thesis we will use the original OS-AP (without including identity tags) as the basis for our investigations (e.g. see section 9).

8.2 Object Independent Authorisation Protocol

Contrary to the OS-AP, the Object Independent Authorisation Protocol (OI-AP) uses multiple authorisation secrets in the same session to access many target objects. The advantage of such a procedure is that it does not require to start a new protocol run for every object that has to be accessed. This relieves the trusted platform from generating and handling more simultaneous sessions. The main specification requires the TPM only to harbour two simultaneous sessions. Thus, if simultaneous access to more then two independent objects would be required, the TPM would have to evoke the $Session - caching$ functionality of the TPM. This would tax the computational capacity for regular operations of the TPM in an infeasible manner.

The protocol itself can be divided in three parts:

1. the start protocol command submission and its feedback
2. the submission of the first command, the generation of the object dependent shared secret, the execution of the command and its feedback
3. the request to continue the session with a command that accesses a new object, the commands execution and its feedback

8.2.1 Description

In this section we will elaborate on the TCPA definition of the OI-AP.

Message 1.	$Owner \rightarrow TPM$	:	TPM_OIAP
Message 2.	$TPM \rightarrow Owner$	:	$authHandle, authlastNonceEven$
Message 3.	$Owner \rightarrow TPM$	:	$tag, paramSize, ordinal, inArgOne, inArgTwo$
			$authHandle, nonceOdd, continueAuthSession$
			$HMAC(key.usageAuth, D_1)$ $where$
			$D_1 = \langle InArgHashOne, InArgHashTwo \rangle$
Message 4.	$TPM \rightarrow Owner$	:	$tag, paramSize, returnCode, outArgOne$
			$nonceEven, continueAuthSession$
			$HMAC(key.usageAuth, D_2)$ $where$
			$D_2 = \langle OutArgHashOne, OutArgHashTwo \rangle$

In message 1 the TPM_OIAP command and its relevant parameters are submitted. On this evocation the TPM generates the session by allocating the required resources and generating an ID for these resources. Afterwards it generates the nonce $authLastNonceEven$. In message 2 the TPM returns the ID for the session ($authHandle$) and the newly generated nonce ($authLastNonceEven$). Thereupon, the initiator generates the nonce $nonceOdd$ and computes $inAuth$ by calculating the HMAC of the authorisation secret of the object ($key.usageAuth$),

the hash sum of the input parameters ($inParamDigest$) and the SHA1 result of the authentication setup parameters ($inAuthSetupParams$). The initiator includes that information in Message 3 to submit the command that accesses the protected object. The TPM verifies the value of $authHandle$, recalculates the HMAC in $inAuth$ and compares it with the newly received value. Afterwards it executes the command, generates the return code and a new nonce to replace the value in $authLastNonceEven$. Further it calculates $resAuth$ by building the HMAC of the authorisation secret of the object, the hash sum of the outgoing parameters ($outParamDigest$) and $outAuthSetupParams$. In message 4 the initiator receives this information and verifies the integrity. It does so by producing the equivalent of the newly received $resAuth$ and compares this with the received value.

The OI-AP uses the same mechanism for continuation of a session as the former OS-AP. Message 3 and 4 carry the information whether or not the session should be aborted after one run. If the value of $continueAuthSession$ is set to $TRUE$ in both messages, the session continues with a slight aberration to message 3 and 4. For the next messages we assume that the owner wants to address another object, hence requires a new authorisation secret ($newkey.usageAuth$).

$$
\begin{aligned}
\text{Message 5.}\quad & Owner \rightarrow TPM \;:\; tag, paramSize, ordinal, inArgOne, inArgTwo \\
& \quad nonceOdd, continueAuthSession \\
& \quad HMAC(newkey.usageAuth, D_3)\ where \\
& \quad D_3 = \langle InArgHashOne, InArgHashTwo \rangle \\
\text{Message 6.}\quad & TPM \rightarrow Owner \;:\; tag, paramSize, returnCode, outArgOne \\
& \quad nonceEven, continueAuthSession \\
& \quad HMAC(newkey.usageAuth, D_4)\ where \\
& \quad D_4 = \langle OutArgHashOne, OutArgHashTwo \rangle
\end{aligned}
$$

$$
\begin{aligned}
&Whereas: \\
&\quad InArgHashOne = SHA1(ordinal, inArgOne, inArgTwo) \\
&\quad InArgHashTwo = authHandle, authLastNonceEven, \\
&\qquad\qquad nonceOdd, continueAuthSession \\
&\quad OutArgHashOne = SHA1(returnCode, ordinal, outArgOne) \\
&\quad OutArgHashTwo = authHandle, nonceEven, nonceOdd, \\
&\qquad\qquad continueAuthSession.
\end{aligned}
$$

The initiator gernerates a new nonce ($nonceOdd$) to replace the old value and generates $inAuth$ by calculating the HMAC with the authorisation secret of the new object and the other updated values. Message 5 is similar to message 3, except it does not contain the value of $authHandle$ and thus the TPM is not verifying the session ID anymore. The rest of the process remains unchanged.

Since we have elaborated on the protocol analysis of streaming protocols in section 8.1 we will omit a full description of the complete analysis.

8.2.2 Discussion

The FDR analysis of our protocol revealed a flaw similar to the one described in our OS-AP analysis. The participants are not properly authenticated to each other. Depending on the command and the scenario the protocol is used in this can cause problems. At this point we will omit a description of an example attack since the structure and solution to the attack is identical to our OS-AP attack. The same holds for the discussion about the seriousness of the flaw.

8.3 Authorization Data Insertion Protocol

The Authorization Data Insertion Protocol (ADIP) binds authorisation data during the creation of an object to the newly generated object.

On creation of a new object the creator has the choice whether or not he wants to include an authorisation secret. The main specification recommends that this option should always been taken, since the authorisation check implicitly verifies the integrity of the I/O parameters. The requirements for a protocol that ensures the proper insertion of an owner-generated secret into an object seem obvious. The main issues are integrity and confidentiality. The integrity originates form the facts that the creator has the choice to verify the authorisation data that has been sent to the TPM and that only a specific TPM can decrypt the information. The confidentiality of the authorisation data stems form a Vernam encryption of the data with some temporarily established session key.

During creation the user must first prove knowledge of the parent authorisation secret. More precisely, the user must establish an OS-AP session for the parent object in order to create an ephemeral secret. This secret is used to ensure the confidentiality of the new authorisation secret.

The protocol can be divided in 4 stages.

1. user chooses a 20 byte authorisation code (for later purposes called Auth.C.) for the object

2. user proves knowledge of the secret that is required to access the parent, by using a previously established OS-AP session (creates the ephemeral secret)

3. user issues the following command:
   $TPMcreationrequest(XOR(Auth.C., ephemeralsecret))$

4. the TPM closes the OS-AP session

8.3.1 Description

The TCPA main specification [TCPA02] defines the ADIP as follows:

Message 1.	$Alice \rightarrow Bob$	:	$TPM\_OSAP, parentHandle, nonceOddOSAP$
Message 2.	$Bob \rightarrow Alice$	:	$authHandle, authLastNonceEven, nonceEvenOSAP$
Message 3.	$Alice \rightarrow Bob$	:	$tag, paramSize, ordinal, inArgOne, inArgTwo$
			$entityAuthData \oplus$
			$SHA1(sharedsecret, authLastNonceEven)$
			$authHandle, nonceOdd, continueAuthSession$
			$HMAC(sharedSecret, D_1)$ *where*
			$D_1 = \langle InArgHashOne, InArgHashTwo \rangle$
Message 4.	$Bob \rightarrow Alice$	:	$tag, paramSize, returnCode, outArgOne$
			$nonceEven, continueAuthSession$
			$HMAC(sharedsecret, D_2)$ *where*
			$D_3 = \langle OutArgHashOne, OutArgHashTwo \rangle$

Whereas :

$sharedsecret = HMAC(parent.usageAuth, nonceEvenOSAP$
$nonceOddOSAP)$
$InArgHashOne = SHA1(ordinal, parentHandle, inArgTwo, newAuth)$
$InArgHashTwo = authLastNonceEven, nonceOdd, continueAuthSession$
$OutArgHashOne = SHA1(returnCode, ordinal, outArgOne)$
$OutArgHashOne = nonceEven, nonceOdd, continueAuthSession.$

The first two messages represent the OS-AP session header to authorise the access of the parent object. In message one the TPM_OSAP command is passed along with all its input parameters, such as the identifier of the parent object and a nonce ($nonceOddOSAP$). On reception of this message the TPM creates the required data structure for the OS-AP session. The session identifier thereafter is called the $authHandle$. It generates two nonces called $authLastNonceEven$ and $authLastNonceEvenOSAP$ and a session key (called $sharedSecret$). It does so by building a single HMAC of the authentication secret of the parent object, the $nonceEvenOSAP$ and the $nonceOddOSAP$. In message two it transfers the session identifier, the $authLastNonceEven$ and $nonceEvenOSAP$ back to the initiator. The initiator reproduces the shared secret calculation and stores all relevant data in its session space. So far everything is equal to the $OS-AP$ protocol. However at this point the $ADIP$ specific part starts. The Initiator generates the nonce that is required for the second handshake, selects the authentication secret for the newly generated object and XORs (Vernam encryption) this value with the result of a hash calculation over the shared secret and the $authLastNonceEven$. The result of this XOR operation is called $newAuth$. Finally it generates $inAuth$ by generating the HMAC of the session key, the input parameters ($inParamDigest$) and parameters that were required to set up the

authentication (called the $inAuthSetupParams$). Afterwards the initiator wraps up the information and sends it to the TPM (message 3). The TPM verifies the value of $authHandle$, loads the $authLastNonceEven$ form its tamper-proof storage and recalculates the value of the received $inAuth$ and compares its result with the received value. Afterwards it decrypts the new authorisation value by using another Vernam encryption over $newAuth$. Finally it performs the command that creates the new object, stores the value obtained by the Vernam decryption in the objects data structure and generates the acknowledgement of the procedure. The integrity of the return parameters is guarantied by an HMAC. This HMAC is generated by hashing the session secret, the return parameters ($outParamDigest$) and $outAuthSetupParams$. On receiving message 4 the initiator verifies the received data by recalculating the HMAC consisting of the session key, $outParamDigest$, $outAuthSetupParams$ and comparing the result with $resAuth$.

The modifications that where necessary to accelerate our *Casper* / FDR investigation are similar to those described in chapter 8.1.

Our investigation spotted that the protocol contained specification violations. Since the verification of this protocol only revealed vulnerabilities that stemmed from the identity flaw in the underlying OS-AP we will not describe the attacks at this point.

8.3.2 Discussion

As mentioned earlier this protocol (model) contains traces that violate our specification, however the basic functionality is still given. The worst that can happen is that the attacker achieves the creation of an identical object that has the same authorisation secret as the original. It seems that the inventors of the protocol wanted to generate security by using as much security enhancing elements as possible. This in one respect may be good, since the weaknesses of the underlying protocol is counterbalanced, however it complicates the protocol by rehashing the same values again and again. At this point we will not elaborate on a suggestion how to reduce the protocol to a minimalist version, that still grants all the advantages without taxing the TPM capacity as it does in the original. This may be an avenue for future research.

8.4 Authorisation Change

During the lifetime of certain objects it may be useful to change the authorisation secret. There are two ways to accomplish that goal: first the Authorization Data Change Protocol (ADCP) and second the more elaborate Asynchronous Authorization Change Protocol (AACP).

8.4.1 Authorization Data Change Protocol

The Authorization Data Change Protocol (ADCP) is used whenever the owner of a protected object requests to change the authorisation secret. Prior to execution of the actual authorisation secret modification two authorisation sessions are required. The first session must be of type OS-AP and the second session can be an OS-AP or OI-AP. In the following investigation we will assume it is an OS-AP.

The first session authorises the access to the parent object of the object that needs a modification of its authorisation secret. This procedure establishes a shared secret. This shared secret will be used as a key for the Vernam encryption of the new authorisation value. The second session verifies whether the requestor has the right to access and change the data of the current object. Clearly there arise two problems:

1. If the object, at which the user wants to change the authorisation data, has no parent key, this standard protocol has to be conducted in a different manner [TCPA02]. In our investigation we do not consider this special case.

2. Since the first session establishes a session key with the authorisation secret of the parent node and this session key is used for the encryption of the new authorisation value everyone who knows the parent authorisation value can decrypt the new authorisation secret of the child object.

The last disadvantage seems to be a minor problem. However if we consider the case where the owner of a system grants a different user the right to store data securely and confidentially on his platform. The user has to be capable to change the authorisation secret in a manner so that it cannot be compromised by the owner of the platform. This is required especially in GRID computing or for developing Digital Rights Management protocols. The protocol itself can be divided in three parts:

1. establishment of the OS-AP session and generation of the session key

2. establishment of the OI-AP session to verify whether the requestor has the right to access the object

3. execution of the command $TPM\_ChangeAuth$ to change the authorisaiton secret

Description

The main specification ([TCPA02]) defines the protocol as follows:

Message 1.	$Alice \rightarrow TPM$	:	$TPM\_OSAP, keyHandle.Parent$
			$nonceOddOSAP.Parent$
Message 2.	$TPM \rightarrow Alice$	:	$authHandle.Parent, authLastNonceEven.Parent$
			$nonceEvenOSAP.Parent$
Message 3.	$Alice \rightarrow TPM$	:	$TPM\_OSAP, keyHandle.entity$
			$nonceOddOSAP.entity$
Message 4.	$TPM \rightarrow Alice$	:	$authHandle.entity, authLastNonceEven.entity$
			$nonceEvenOSAP.entity$

The first four messages are to establish the authorisation sessions between the parent node and the object where the user wants to change the authorisation secret. Since we already described the OS-AP in depth we will not elaborate on them further. We start the description of the protocol by explaining how the shared secret and the value $NewAuth$ are calculated. After finishing the first two handshakes the TPM and the initiator calculate the shared secret (key) by creating the HMAC of the authentication secret of the parent node ($usageAuth.Parent$) and the two nonces $nonceEvenOSAP.Parent$ and $nonceOddOSAP.Parent$. The initiator uses the shared secret as a key for Vernam-encrypting the new authentication value. More precisely it concatenates the values $authLastNonceEven.Parent$ with $sharedSecret$, calculates the SHA1 hash sum of it and XORs the result with the new authorisation value $entityNewAuthData$.

Message 5.	$Alice \rightarrow TPM$	:	$tag, paramSize, ordinal, keyHandle.Parent$
			$protocolID, NewAuth, entityType, encDataSize$
			$encData, authHandle.Parent$
			$nonceOdd.Parent, continueAuthSession$
			$HMAC(key.UsageAuth.Parent, D_1)$ $where$
			$D_1 = \langle InParamDigest, inAuthSetupParams \rangle$
			$entityAuthHandle, entityNonceOdd$
			$continueEntitySession$
			$HMAC(key.UsageAuth.Entity, D_2)$ $where$
			$D_2 = \langle InParamDigest, inAuthSetupParams2 \rangle$

Message 6. $TPM \rightarrow Alice$: $tag, paramSize, returnCode, outDataSize, outData$
$nonceEven.Parent, continueAuthSession$
$HMAC(key.usageAuth.Parent, D_3)$ *where*
$D_3 = \langle OutParamDigest, OutAuthSetupParams \rangle$
$nonceEven.Entity, continueAuthSession$
$HMAC(usageAuth.Entity, D_4)$ *where*
$D_4 = \langle OutParamDigest, OutAuthSetupParams2 \rangle$

Whereas :
$InParamDigest = SHA1(ordinal, protocolID, newAuth, entityType$
$encDataSize, encData)$
$inAuthSetupParams = authHandle.Parent, authLastNonceEven.Parent$
$nonceOdd.Parent, continueAuthSession$
$inAuthSetupParams2 = AuthHandle.Entity, entitylastNonceEven$
$entityNonceOdd, continueEntitySession$
$outParamDigest = SHA1(returnCode, ordinal, outDataSize, outData)$
$outAuthSetupParams = authHandle.Parent, nonceEven.Parent$
$nonceOdd.Parent, continueAuthSession$
$outAuthSetupParams2 = authHandle.Entity, nonceEven.Entity$
$nonceOdd.Entity, continueEntityAuthSession.$

In message 5a to 5f the initiator submits the $TPM\_ChangeAuth$ command with all its parameters. The TPM verifies the legitimacy of this call. It does so by loading the internally stored corresponding values and reverses the operations that were performed by the initiator to produce the value $inAuthSetupParams$. This also guaranties that the data is fresh, since nonces are included, and ensures the integrity of $NewAuth$ and $encData$ (the important parameters). Afterwards the TPM extracts the new authorisation value, decrypts the object, changes the authorisation value of the object and encrypts the modified object again (now called $outData$). After returning the output parameters of the $TPM\_ChangeAuth$ command the TPM enforces that both session OS-AP and OI-AP / OS-AP are terminated. In message 6 the initiator receives the feedback of the authorisation value modification.

Discussion

After converting the protocol to a *Casper* readable format we used FDR to verify whether attacks are possible. As in the protocols before, FDR showed that the OS-AP flaw was not rectified by the high level protocol. However, the revealed specification violations are inconsequential for the overall functionality of the protocol. The ADCP transmits, in message 3 and 4, the complete encrypted object that should be altered. In the worst case, if the initiator of the protocol assumes that the command was not executed he simply retransmits the original

object. Hence the command will again be executed upon the initial object. Side-effects that can occur while unintentionally executing twice the same command upon the same piece of data cannot occur.

Returning to the, initially mentioned, problems of this protocol we can conclude that this protocol is useless for GRID computing or DRM protocols. The key for the encryption of the new authorisation value of the object is generated form the authorisation secret of the parent node. Thus the owner of the parent object can overhear the transaction and encrypt the new value. This grants him continuous access to the object. TCPA has designed, specifically to circumvent this attack possibility, a protocol that allows an asynchronous authorisation change. It enables the owner of the object, with the permission of the parent object owner, to exclude the parent object owner.

8.4.2 Asymmetric Authorization Change Protocol

The Asymmetric Authorization Change Protocol (AACP) allows the owner of the platform (or owner of parent node) and the owner of a local object to agree on an asynchronous change of the authorisation secret. This change has to be initiated by the owner of the parent node. After completion of the transaction the parent is unable to inspect the content of the object. However he is still in control, so for instance (important for DRM) he still can erase the object.

First an OI-AP session has to be initiated, afterwards the run is completed by the following two commands $TPM\_ChangeAuthAsymStart$ and $TPM\_ChangeAuthAsymFinish$.

The protocol can be divided into five stages:

1. establishment of an OI-AP session and initiation of the $TPM\_ChangeAuthAsymStart$ command by the owner of the parent node
2. generation of a temporary asymmetric key pair and transmission of the public part as well as a proof that the private part is only known to the TPM (non-migratable key attestation) to the requestor, which in this regard is the owner.
3. the owner of the parent node forwards to the owner of the child object the information provided by step 2 (key + proof)
4. the owner encrypts the new authorisation value with the temporary public key provided by the TPM and passes the data on to the owner
5. the owner submits the $TPM\_ChangeAuthAsymFinish$ command and the TPM decrypts the value of the new authorisation secret and fulfils the changes accordingly

Since certain parts of the protocol follow the same patterns as the protocols described above we will only elaborate on the new parts.

Description

The AACP is defined as follows:

$$
\begin{aligned}
&\text{Message 1.} \quad Alice \rightarrow TPM \;:\; TPM\_OIAP \\
&\text{Message 2.} \quad TPM \rightarrow Alice \;:\; authHandle.IDKey, authlastNonceEven \\
&\text{Message 3.} \quad Alice \rightarrow TPM \;:\; tag, paramSize, ordinal, idHandle \\
&\qquad\qquad\qquad\qquad\quad antiReply.nonce, tempKey, authHandle.IDKey \\
&\qquad\qquad\qquad\qquad\quad nonceOdd, continueAuthSession \\
&\qquad\qquad\qquad\qquad\quad HMAC(IDKey.usageAuth, D_1)\ where \\
&\qquad\qquad\qquad\qquad\quad D_1 = \langle inParamDigest1, inAuthSetupParams1 \rangle
\end{aligned}
$$

$$
\begin{aligned}
&Whereas: \\
&\quad inParamDigest1 = SHA1(ordinal, antiReply.nonce, tempKey) \\
&\quad inAuthSetupParams1 = authHandle.IDKey, authLastNonceEven \\
&\qquad\qquad\qquad\qquad nonceOdd, continueAuthSession.
\end{aligned}
$$

As mentioned before the command requires that an OI-AP session is established prior to execution to grant access to the identity key of the TPM. The first two messages are dealing with the establishment of the OI-AP session; since we have described the OI-AP protocol in great detail, we will not elaborate further on the first two messages. In Message 3 the owner of the parent node uses the information form the OI-AP session to generate the required input for the $TPM\_ChangeAuthAsymStart$ command. He includes, among various other parameters, the identifier of the ID key ($idHandle$), a nonce for anti reply purposes, he stores the parameters for the public key generation in $tempKey$ and the data that ensures the freshness and integrity of the whole message in $idAuth$. The field $tempKey$ contains all necessary information to generate a public key pair; one example for such a parameter would be the required key length. The value $idAuth$ is calculated by an HMAC operation that uses the ID Key authorisation secret as its key and $inParamDigest$ and $inAuthSetupParams$ as its data values.

$$
\begin{aligned}
&\text{Message 4.} \quad TPM \rightarrow Alice \;:\; tag, paramSize, returnCode, certifyInfo \\
&\qquad\qquad\qquad\qquad\quad sigSize, sig, ephHandle, tempKey \\
&\qquad\qquad\qquad\qquad\quad nonceEven, continueAuthSession \\
&\qquad\qquad\qquad\qquad\quad HMAC(IDKey.usageAuth, D_2)\ where \\
&\qquad\qquad\qquad\qquad\quad D_2 = \langle outParamDigest1, outAuthSetupParams1 \rangle
\end{aligned}
$$

Message 5a. $Alice \rightarrow TPM$: $tag, paramSize, ordinal, parentHandle, ephHandle$
$entityType, newAuthLink, newAuthSize$
$encNewAuth, encDataSize, encData$
$authHandle.Parent, nonceOdd, continueAuthSession$
$HMAC(parentKey.usageAuth, D_3)$ *where*
$D_3 = \langle parentHandle, inParamDigest2,$
$inAuthSetupParams2 \rangle$

Message 6a. $Bob \rightarrow Alice$: $tag, paramSize, returnCode, outDataSize, outData$
$saltNonce, changeProof, nonceEven$
$continueAuthSession$
$HMAC(parentKey.usageAuth, D_4)$ *where*
$D_4 = \langle outParamDigest2, outAuthSetupParams2 \rangle$

Whereas :

$outParamDigest1 = SHA1(returnCode, ordinal, certifyInfo, sigSize$
$sig, ephHandle, tempKey)$
$outAuthSetupParams1 = authHandle.IDKey, nonceEven, nonceOdd$
$continueAuthSession$
$inParamDigest2 = SHA1(ordinal, entityType, newAuthLink, newAuthSize$
$encNewAuth, encDataSize, encData)$
$inAuthSetupParams2 = authHandle.Parent, authLastNonceEven$
$nonceOdd, continueAuthSession$
$outParamDigest2 = SHA1(returnCode, ordinal, outDataSize, outData$
$saltNonce, changeProof)$
$outAuthSetupParams2 = authHandle.ParentKey, nonceEven, nonceOdd$
$continueAuthSession.$

Once the TPM has received message 3 it verifies the authorisation value of the ID Key (identified by value $idHandle$) and the parameters that initialise the public-key-pair generation algorithm. If these parameters evaluate in a positive manner the TPM produces the public / private key pair and assigns the name $ephHandle$ to them. Afterwards it stores the public key part in $tempKey$ and produces a certification of the newly generated key pair. It is not clear whether the TPM uses the command $TPM\_CertifyKey$ to do so, however it seems very likely; For anti reply purposes the nonce $antiRepy$ is included. This certificate is stored in the $certifyInfo$ field. Thereupon $certifyInfo$ is signed with the ID key that is referenced by $idHandle$ and the result of this signing process is stored in sig.

After this process the TPM submits message 4. On receiving message 4, the user verifies the freshness and the integrity of message 4 with the standard operations, which were described in the earlier protocols (HMAC and SHA1 re-calculations). At this point the specification weakens. It does not describe how

the owner of the identity key forwards the relevant information such as the public key to the owner of the child object. In our later investigation we will discuss one approach (see chapter 11). For the initial description it is only important that the owner of the child node encrypts the new authorisation value with the public potion of $tempKey$ (now called $encNewAuth$). This value is passed on to the owner of the ID key and he submits, in message 5, the $TPM\_ChangeAuthAsymFinish$ command that closes the transaction.

The TPM checks whether the $authHandle$ parameter grants access to the ID key. Afterwards it decrypts the value $endData$ (the object that has to be modified) and uses the $tempkey.private$ to decrypt $encNewAuth$. Then it calculates the HMAC of the decrypted new authorisation secret ($newAuthSecret$) whereby it uses the old authorisation secret of the target object as a key (stored in $encData.currentAuth$). This HMAC is compared with the parameter $newAuthLink$ to ensure integrity. After this operation the old authorisation secret is replaced by the new value and the complete object is encrypted with the key that is referenced by $parentHandle$. Finally the TPM produces a nonce ($saltNonce$), to increase the entropy of the $changeProof$ value. Then it calculates the $changeProof$ certificate to testify that the desired authorisation value modification took place. It does so by generating the HMAC of the concatenation of the $saltNonce$ and the value $noceOdd$ and uses the new authorisation secret as a key ($newAuthSecret$). In message 6 all required data is transferred to the owner of the parent object.

This is the second place where the specification is fuzzy. It is not clear how and, more important, what information is passed to the owner of the child object. At least he has to receive the value $changeProof$ to determine that the transaction was successful.

Discussion

Our analysis showed that this protocol still carries the underlying identity attack. Since this protocol is a derivative of the ADCP (see chapter 8.4.1) and the complete objects that have to be altered are transmitted, the attack remains without serious consequences.

However, we only tested the two-way communication between the owner and the TPM. [TCPA02] mentions only in a strange way how the real protocol should look. In our DRM chapter 11 we will expand this protocol so that it covers as well the communication between the owner and the guest.

8.5 Conclusion

In this chapter we have shown how the basic protocols of the TCPA work. We performed a thorough analysis of all five protocols. We described the complete

analysis technique at the OS-AP. We did so by pruning away all unnecessary fields and modelling the subsequent behaviour in CSP. Afterwards we restricted the ranges of the nonces and the command input and output types in such a way, that it was possible for the initiator and the responder to have access to a source of unbounded supply of nonces. We argued that our transformations were sensible and employed a simplification theory developed by Lowe and Hui to ensure that these abstractions did not introduce false-negatives. Additionally we used the data independence theory developed by Roscoe to set our nonce and command input / output recycling algorithm on formal grounds.

Our analysis spotted various attacks upon OS-AP and OI-AP. All of these were based on the same principle. The intruder could redirect the messages and therefore pretend to be another person. The claim that OS-AP and OI-AP can be used to establish a proper authentication session [TCPA02] does not hold. This flaw can easily be rectified by including the sender and receiver identities in the message and in the accompanying hash sums. It is important to say that only messages 3 and 4 have to be altered. All other messages could remain unchanged. This is due to the rolling nonce paradigm[1].

As we have discussed in the later part of this chapter, this flaw does not hinder ADIP, ADCP and AACP to work according to the TCPA specification. They do so by comunicating additional information, such as the complete encrypted object that has to be altered.

[1]The rolling nonce paradigm requires that a nonce from one side can only be used for one message and its reply, after that it has to be exchanged with a new value.

Chapter 9

Session Caching

The TCPA specification requires the TPM to be able to provide internal space for only two authorisation sessions. However, it may very well be that more parallel sessions are required. Therefore, a mechanism is provided to temporarily suspend idle sessions and store their state information externally. The TCPA calls this process authorisation context management. The specification defines two commands, called $TPM\_SaveAuthContext$ and $TPM\_LoadAuthContext$ to suspend or resume a session. These commands create or receive a blob that contains the required information about the session. A so-called external session manager handles the external storage and the retrieval of the data.

The specification claims that this process is not required to be trusted. In this section we will explore whether or not this statement holds. We will generate a CSP model of all processes that are directly involved as well as certain meta-processes that are required to ensure a proper command flow. First we will describe the complete process as it appears in the real world. Due to the strict state space limitations, we will use the abstracted version of the OS-AP (see section 8) to establish communication between the TPM and the user. Additionally, we prune away all features that are not directly necessary for the context management. The resulting model confirmed the need for strict verification procedures that can determine whether or not the cached session information has been altered. This section will conclude with a discussion about our results and a generalisation of certain abstraction techniques in such a setting.

9.1 The real world model

As mentioned above, we will produce a model that maps the system of the real world as accurately as possible. This system contains the following processes: An owner that will launch an OS-AP session; an intruder that can intercept every message that was sent to and from the owner to the TPM and can interact with the external session manager; and the TPM with all its relevant sub- and meta

processes and an external data manager. Figure 9.1 shows the general overview of the network.

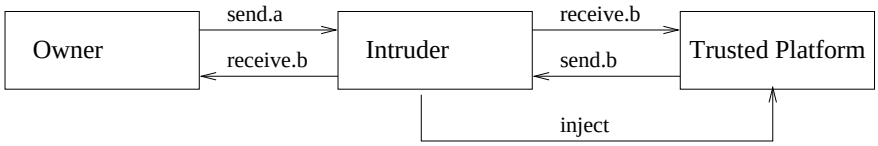

Figure 9.1: Session caching

We will only focus on the establishment of an OS-AP session; we are fully aware that there may be other implications if we would consider other protocols such as the AACP; to inspect more protocols lies beyond the scope of this thesis. However, at the end of this chapter we will present techniques that ease the development of the necessary models.

Generally, the TPM can accept as many session requests as it receives. Once two sessions are already active, the authorisation session management launches. It is important to note that this management process is *not* within the TPM. Only one command that produces encrypted blobs, which contain all the relevant information to resume the session, and another command to load these blobs to resume the session are embedded in the TPM. Hence, the overall supervision lies outside the TPM and cannot be trusted as a result. The general process of session suspension and resumption is as follows:

1. An external process receives the request that a user wants to establish a third session with the TPM. Thereupon, a $TPM\_SaveAuthContext$ command is sent to the TPM.

2. The TPM gathers the necessary information such as the *authorisation handle*, *nonce*, *digest* and the $ephemeralsecret$ and stores the information in a specific data structure.

3. This data structure is encrypted by a public key of the TPM. The private portion of this public-key pair is non-migratable. The command returns the encrypted blob to the caller.

4. Finally, the external process stores the blob.

5. Once the session has to be resumed, the external process collects all relevant information to issue the $TPM\_LoadAuthContext$ command.

6. The $TPM\_LoadAuthContext$ command decrypts the blob and stores the information within the internal session storage.

We tried to model the whole system as close to the specification as possible. All events that are explicitly stated in the specification are named with the prefix *int_* for internal. In addition to these events, we had to introduce events, such as addressing of memory resources, that were not stated in the main specification. These events have the prefix *meta_*.

9.1.1 The TPM context management

The TPM context management process contains the two commands that enable the session caching process. The overall process is very simple and consists of two interleaved processes ($TPM\_LoadAuthContext$ and $TPM\_SaveAuthContext$).

$$
\begin{aligned}
&TPMContextManagement = \\
&\quad TPM\_SaveAuthContext \;|||\; TPM\_LoadAuthContext
\end{aligned}
$$

The TPM_SaveAuthContext command receives an authorisation session handle and collects the information about the session. It stores all relevant information in a data structure. The specification demands that this session blob may only be encrypted by the originating TPM — hence, we assume that this structure will be encrypted with the public part of a non-migratable key. Furthermore, it states that this blob shall contain information that can be used to verify the integrity of the object. Additionally, every blob is also tied to a specific TPM via the value TPM_Proof. Hence, the TPM can always determine whether or not it has generated a specific blob. We will further discuss this TPM_Proof value in our results section. After generating the blob, the TPM releases all internal resources that are linked to the session. This behaviour can be described by the following CSP process:

$$
\begin{aligned}
&TPM\_SaveAuthContext = \\
&\quad int\_startTPM\_SaveAuthContext?authHandle \rightarrow \\
&\quad meta\_readSessionSpace!1?authHandle1 \rightarrow \\
&\quad meta\_readSessionSpace!2?authHandle2 \rightarrow \\
&\quad \text{if } authHandle1 \neq authHandle \wedge authHandle2 \neq authHandle \\
&\quad \text{then } int\_TPM\_SaveAuthContextFailureWrongAuthHandle \rightarrow \\
&\qquad TPM\_SaveAuthContext \\
&\quad \text{else } meta\_readSessionParallel!authHandle?nonce?digest?Secret \rightarrow \\
&\qquad int\_ResetSession!authHandle \rightarrow meta\_decreaseSessionCounter \rightarrow \\
&\qquad int\_finishTPM\_SaveAuthContext!(authHandle, nonce, digest, Secret) \rightarrow \\
&\qquad TPM\_SaveAuthContext
\end{aligned}
$$

We have divided the $TPM\_SaveAuthContext$ command into two commands, a start command that receives the input parameters (called $int\_startTPM\_SaveAuthContext$) and the finish command that returns the output of the operation (called $int\_finishTPM\_SaveAuthContext$). This method not only

ensures atomicity of the command but also increases the readability. The $int\_startTPM\_SaveAuthContext$ command receives four input parameters: $TCPA\_TAG$, $paramSize$ (which contains the length of the message in bytes), $ordinal$ (the identification number of the TPM command) and the $authHandle$ (the identifier of the session that has to be suspended). After receiving the input parameters, the TPM verifies whether or not the session (identified by $authHandle$) is indeed active. It does so with the event $meta\_readSessionSpace$. This command accepts an address that points to a specific location within the TPM's session space and receives the identifier of the session that is stored in the memory space. If the session in question is not active, an error message will be raised. If the relevant information is available $meta\_readSessionParallel$ retrieves the necessary data. After that command is given, all resources are freed ($int\_ResetSession$ and $meta\_decreaseSessionCounter$). The channel $int\_finishTPM\_SaveAuthContext$ returns the $TCPA\_TAG$, the length of the output in bytes ($paramSize$), the return code of the command ($returnCode$), the size of the blob ($authContextSize$) and the blob itself ($authContextBlob$). We deliberately omit the encryption and the creation of hash sums to guarantee the integrity of the data. We will control the desired behaviour by implementing an intruder that cannot overhear the session blobs or cannot alter session blobs.

The TPM_LoadAuthContext command is responsible for resuming a suspended session. It receives a session blob, decrypts the blob, verifies its integrity and stores the obtained information within the internal session memory. Afterwards the command returns the authorisation handle and the session can be resumed. The following CSP description depicts this operation:

$TPM\_LoadAuthContext =$
$int\_startTPM\_LoadAuthContext?Blob \rightarrow$
$meta\_readSessionCounter?counter \rightarrow$
if $counter > 0$
then $int\_TPM\_LoadAuthContextFailureNoSessionSpaceAvailable \rightarrow$
$TPM\_LoadAuthContext$
else $meta\_writeSessionParallel!N!fst(Blob)!sec(Blob)!rd(Blob)!fourth(Blob) \rightarrow$
$meta\_increaseSessionCounter \rightarrow$
$int\_finishTPM\_LoadAuthContext!first(Blob) \rightarrow$
$TPM\_LoadAuthContext$

As with the latter command we distinguish between a start and a finish command. The $int\_startTPM\_LoadAuthContext$ channel communicates all relevant input parameters: the $TCPA\_TAG$, the parameter size ($paramSize$), ordinal ($ordinal$), the size of the blob ($authContextSize$) and the blob itself ($Blob$). Thereupon, the TPM verifies whether or not memory is available to store the required information ($meta\_readSessionCounter$). If this test reveals that two sessions are currently active an error message will be issued ($int\_TPM\_LoadAuthContextFailtureNoSessionSpaceAvailable$). If there is at least one free

session space, the command *meta_writeSessionParallel* stores the state information in the available memory slot and increases the number of active sessions (*meta_increaseSessionCounter*). Lastly, the process concludes by returning the authorisation handle (*int_finishTPM_LoadAuthContext*) to the caller. This channel also returns the $TCPA\_TAG$, the length of the message (*paramSize*) and the return code (*returnCode*).

9.1.2 The internal session data storage

The internal session storage contains the authorisation handle, the current nonce, a digest, which in most of the cases is a pointer that addresses a particular object, and the authorisation secret (ephermal secret). The natural and preferable CSP solution for FDR would be:

$$
\begin{aligned}
&TPMOwnMem(authHandle, nonce, digest, ephSecret) = \\
&\quad TPMOwnerMemStor(1, N) \;|||\; TPMOwnerMemStor(2, N) \;|||\; \\
&\quad TPMOwnerMemStor(3, N) \;|||\; TPMOwnerMemStor(4, N)
\end{aligned}
$$

$$
\begin{aligned}
&TPMOwnerMemStor(name, data) = \\
&\quad int\_setTPMOwnerMemStor!name?newData \rightarrow \\
&\quad TPMOwnerMemStor(name, newData) \\
&\quad \Box \\
&\quad int\_readTPMOwnerMemStor!name!data \rightarrow \\
&\quad TPMOwnerMemStor(name, data)
\end{aligned}
$$

One memory block consists of four interleaved processes and each process holds a type identifier and the data value. Every sub-process offers events that allow read (*int_readTPMOwnerMemStor*) and write (*int_writeTPMOwnerMemStor*) operations on the data values. Unfortunately, this solution would increase the complexity of the overall system dramatically. Therefore, we have only one process for each memory block:

$$
\begin{aligned}
&Session = \\
&\quad Session'(1, N, N, N, N, 0) \\
&\quad [|\{|meta\_readSessionCounter, int\_ResetSessions, \\
&\quad meta\_increaseSessionCounter, meta\_decreaseSessionCounter|\}|] \\
&\quad Session'(2, N, N, N, N, 0)
\end{aligned}
$$

Both $Session'$ processes synchronise on certain meta-events. *meta_readSessionCounter*, *meta_increaseSessionCounter* and *meta_decreaseSessionCounter* are responsible for keeping track of the number of active sessions and for providing this state information to requesting processes. The internal event *int_ResetSessions* is raised at the beginning of the boot-cycle of the trusted platform and initialises the memory with predefined values.

The session storage itself is represented by one process that is twice instantiated in a different manner. We omitted all internal commands that are not required for the analysis of the authorisation caching mechanism.

$$
\begin{aligned}
&Session'(space, authHandle, nonce, digest, ephSecret, count) = \\
&\quad meta\_readSessionParallel!authHandle!nonce!digest!ephSecret \rightarrow \\
&\quad Session'(space, authHandle, nonce, digest, ephSecret, count) \\
&\quad \Box \\
&\quad meta\_writeSessionParallel!authHandle?nHandle?nnonce?ndigest?nSecret \rightarrow \\
&\quad Session'(space, nHandle, nnonce, ndigest, nSecret, count) \\
&\quad \Box \\
&\quad meta\_readSessionParallel2!space!authHandle!nonce!digest!ephSecret \rightarrow \\
&\quad Session'(space, authHandle, nonce, digest, ephSecret, count) \\
&\quad \Box \\
&\quad meta\_writeSessionParallel2!space?nHandle?nnonce?ndigest?nSecret \rightarrow \\
&\quad Session'(space, nHandle, nonce, ndigest, nSecret, count) \\
&\quad \Box \\
&\quad meta\_readSessionCounter!count \rightarrow \\
&\quad Session'(space, authHandle, nonce, digest, ephSecret, count) \\
&\quad \Box \\
&\quad meta\_readSessionSpace!space?authHandle \rightarrow \\
&\quad Session'(space, authHandle, nonce, digest, ephSecret, count) \\
&\quad \Box \\
&\quad meta\_increaseSessionCounter \rightarrow \\
&\quad \text{if } count > 1 \\
&\quad \text{then } Session'(space, authHandle, nonce, digest, ephSecret, count) \\
&\quad \text{else } Session'(space, authHandle, nonce, digest, ephSecret, count + 1) \\
&\quad \Box \\
&\quad meta\_decreaseSessionCounter \rightarrow \\
&\quad \text{if } count = 0 \\
&\quad \text{then } Session'(space, authHandle, nonce, digest, ephSecret, count) \\
&\quad \text{else } Session'(space, authHandle, nonce, digest, ephSecret, count - 1) \\
&\quad \Box \\
&\quad int\_ResetSessions \rightarrow Session'(space, N, N, N, N, 0) \\
&\quad \Box \\
&\quad int\_ResetSession!authHandle \rightarrow Session'(space, N, N, N, N, 0)
\end{aligned}
$$

The process $Session'$ is parameterised with a memory block identifier, the required information about the state of the session and the overall session counter. The real world process would offer events to alter or read the session state information and the session counter. Instead of the standard way of reading every bit of information in a single command, we introduced events that read and store the required data in one event ($meta\_readSessionParallel$ and $meta\_writeSessionParallel$). These events address the memory block by data value, whereas the channels $meta\_readSessionParallel$ and $meta\_writeSessionParallel$ store or read the information in the defined memory space. Finally, we provide func-

tionality for two reset operations: one that only resets a particular memory block ($int\_ResetSession$); and one that resets all sessions $int\_ResetSessions$.

During the development of the session caching mechanism we discovered that certain combinations of commands occurred frequently. Thus, in order to reduce the complexity of the CSP model, we designed small processes that represent these command sequences. One of those meta-processes is the $MetaSessionStorage$.

The MetaSessionStorage process collects the state information of the two memory blocks.

$$
\begin{aligned}
&MetaSessionStorage = \\
&\quad meta\_startsessioncount \rightarrow meta\_readSessionCounter?spaces \rightarrow \\
&\quad meta\_readSessionSpace!1?authHandle1 \rightarrow \\
&\quad meta\_readSessionSpace!2?authHandle2 \rightarrow \\
&\quad meta\_finishsessioncount!spaces!authHandle1!authHandle2 \rightarrow \\
&\quad MetaSessionStorage
\end{aligned}
$$

The event $meta\_startsessioncount$ triggers the procedure. First, it determines how many sessions are currently active ($meta\_readSessionCounter$) and it then enquires about the authorisation handles of the stored sessions ($meta\_readSessionSpace$). The event $meta\_finishsessioncount$ conflates the obtained information and communicates it back to the requestor.

9.1.3 The TPM

The process TPM represents the main part of the trusted platform. It consists of four sub-processes, each handling a different state in the OS-AP. For further particulars of the OS-AP see section 8.1. In this model we externalised operations as much as possible. Thus, we have introduced various meta-processes (e.g. $MetaTPMSessionManager$).

The TPM process receives a request for an OS-AP authorisation session, allocates the required resources and stores the information in the allocated memory.

The following CSP process models this behaviour:

$$
\begin{aligned}
&TPM(A, B, tpmcount) = \\
&\quad \Box\, keyHandle : TkeyHandle \bullet \Box\, nonceOddOSAP : NonceOwner \bullet \\
&\quad input.A.B.Message1 \rightarrow \\
&\quad meta\_locksessionscheduler \rightarrow \\
&\quad int\_getnewAuthHandle?aH \rightarrow int\_loadObject!keyHandle?secret \rightarrow \\
&\quad meta\_getNonceNonceManager!A?nonceEvenOSAP \rightarrow \\
&\quad meta\_calculateSharedSessionSecret?sessionsecret \rightarrow \\
&\quad meta\_startstoreAuthHandle!aH!nonceOddOSAP!keyHandle!sessionsecret \rightarrow \\
&\quad meta\_finishstoreAuthHandle \rightarrow \\
&\quad meta\_unlocksessionscheduler \rightarrow \\
&\quad (TPM(A, B, tpmcount + 1) \;|||\; TPM'(A, B, aH))
\end{aligned}
$$

The process is parameterised by the names of the participants, in our case the $Owner$ and the TPM, and a numerical value that keeps track of the number of recently spawned authorisation sessions. Since we explained the OS-A protocol in detail in section 8.1, we project the data values of the four messages on one label (e.g. $Message1$, $Message2$ etc.) and do not discuss their content further. Initially, the process receives the authorisation session request ($receiveOSAP$), and then it notifies the session scheduler that it will perform internal transactions ($meta\_locksessionscheduler$). It requests a new nonce ($int\_getnewNonce$). Note, that since we only focus on the caching mechanism, we omitted the random number generator that generates this particular nonce value.

Afterwards, it loads the object that is addressed by the value within the $keyHandle$ field of OS-AP message 1 from the protected storage and retrieves the authorisation secret that is necessary to access this object. The $meta\_startstoreAuthHandle$ and $meta\_finishstoreAuthHandle$ events allocate not only the required memory, but also store the authorisation handle, the nonce that was sent by the owner, the key handle and the session secret (*HMAC(secret, nonceOddOSAP, nonceEvenOSAP)*) in that memory block. Later we will discuss in more detail what operations are necessary to achieve this transaction. After completing the internal operations, $meta\_unlocksessionscheduler$ ensures that other sessions can process their internal commands. Finally, the TPM process can receive another OS-AP request or proceed with the current session. Note that since the two possesses are interleaved, the option of launching another OS-AP session remains open until the system terminates.

The TPM' process represents the second stage in the protocol.

$TPM'(A, B, cauthhandle) =$
$meta\_locksessionscheduler \rightarrow$
$meta\_startrequestAuthHandle!cauthhandle \rightarrow$
$meta\_finishrequestAuthHandle!cauthhandle?nonce?keyHandle?secret \rightarrow$
$meta\_getNonceNonceManager!A?authLastNonceEven \rightarrow$
$output.B.A.Message2 \rightarrow$
$meta\_unlocksessionscheduler \rightarrow$
$TPM''(A, B, cauthhandle, authLastNonceEven))$

As in the previous stage, the session scheduler events prevent other sessions for executing internal operations. The events $meta\_startrequestAuthHandle$ and $meta\_startstoreAuthHandle$ ensure that the session with the name $authHandle$ is stored within the internal session memory and they return all relevant information about the state of the session. Afterwards, one new nonce is generated ($authLastNonceEven$) and the required information is collected within message 2. The $sendOSAP$ sends message 2 to the $Owner$.

The TPM'' process is built upon the same structure as TPM'. It locks the session scheduler, receives OS-AP message 3 and extracts the relevant information.

$TPM''(A, B, cauthhandle, authLastNonceEven) =$
$meta\_locksessionscheduler \rightarrow$
$\square\ nonceOdd : Nonce \bullet$
$\square\ secret : Tsecret \bullet$
$\square\ nonceOdd : NonceOwner \bullet$
$input.A.B.Message3 \rightarrow$
$meta\_startrequestAuthHandle!cauthhandle \rightarrow$
$meta\_finishrequestAuthHandle!cauthhandle?nonce?keyHandle?stsecret \rightarrow$
$meta\_startstoreAuthHandle!cauthhandle!nonceOdd!keyHandle!stsecret \rightarrow$
$meta\_finishstoreAuthHandle \rightarrow$
if $stsecret = secret$
then $meta\_TPMGrantAccessto!keyHandle!A \rightarrow$
$meta\_unlocksessionscheduler \rightarrow$
$TPM'''(A, B, cauthhandle)$
else $meta\_TPMAccessRejected!keyHandle!A \rightarrow$
$meta\_unlocksessionscheduler \rightarrow$
$TPM'''(A, B, cauthhandle)$

The $meta\_startrequestAuthHandle$ and $meta\_finishrequestAuthHandle$ access the internal session memory to obtain the relevant information about the state of the internal memory. $meta\_startstoreAuthHandle$ and $meta\_finishstoreAuthHandle$ save the updated information about the authorisation session.

Thereupon, the TPM verifies whether or not the ephermal secret, stored in the session memory, is equal to the recently received secret. If the secrets are equal $meta\_TPMGrantAccessto$ indicates that the owner has successfully accessed the object, if not, $meta\_TPMAccessRejected$ will be communicated.

The TPM''' finishes the current OS-AP session. It locks the session scheduler, retrieves the state information of the relevant session and releases the resources that are linked to the authorisation handle ($meta\_ResetSession$). Lastly, it unlocks the session scheduler and sends OS-AP message 4 to the $Owner$.

$$
\begin{aligned}
&TPM'''(A, B, cauthhandle) = \\
&\quad meta\_locksessionscheduler \rightarrow meta\_startrequestAuthHandle!cauthhandle \rightarrow \\
&\quad meta\_finishrequestAuthHandle!cauthhandle?nonceOdd?keyHandle?secret \rightarrow \\
&\quad int\_ResetSession!cauthhandle \rightarrow \\
&\quad meta\_unlocksessionscheduler \rightarrow \\
&\quad output.B.A.Message4 \rightarrow \\
&\quad SKIP
\end{aligned}
$$

The session scheduler guarantees the proper ordering of the various OS-AP runs.

$$
\begin{aligned}
&SessionScheduler = \\
&\quad meta\_locksessionscheduler \rightarrow \\
&\quad meta\_unlocksessionscheduler \rightarrow SessionScheduler
\end{aligned}
$$

It only consists of the $meta\_locksessionscheduler$ and $meta\_unlocksession-scheduler$ events.

9.1.4 The session management meta processes

The process $MetaTPMSessionManager$ is responsible for ensuring that a certain authorisation session is active and that a newly generated session can be stored in a free memory block. The process is further divided into two subprocesses. These two processes do not communicate with each other — thus, they are interleaved.

$$
\begin{aligned}
&MetaTPMSessionManager = \\
&\quad MetaTPMSessionManager' \,|||\, \\
&\quad MetaTPMSessionManager''
\end{aligned}
$$

$MetaTPMSessionManager'$ contains all operations that are necessary to activate a certain authorisation handle.

$MetaTPMSessionManager' =$
$meta\_startrequestAuthHandle?cauthhandle \rightarrow meta\_startsessioncount \rightarrow$
$meta\_finishsessioncount?spaces?authHandle1?authHandle2 \rightarrow$
if $cauthhandle = authHandle1 \vee cauthhandle = authHandle2$
then $meta\_readSessionParallel!cauthhandle?nonce?keyHandle?secret \rightarrow$
$meta\_finishrequestAuthHandle!cauthhandle!nonce!keyHandle!secret \rightarrow$
$MetaTPMSessionManager'$
else $meta\_startTPM\_LoadAuthContext!cauthhandle \rightarrow$
$meta\_finishTPM\_LoadAuthContext \rightarrow$
$meta\_readSessionParallel!cauthhandle?nonce?keyHandle?secret \rightarrow$
$meta\_finishrequestAuthHandle!cauthhandle!nonce!keyHandle!secret \rightarrow$
$MetaTPMSessionManager'$

The initial event $meta\_startrequestAuthHandle$ receives the authorisation handle that has to be activated. $meta\_startsessioncounter$ and $meta\_finishsessioncounter$ inform the process whether or not the session in question is already active. If the session is stored in the internal memory, and therefore usable, $int\_readSessionParallel$ reads the complete information from of the internal session storage and communicates this information back to the caller ($meta\_finishrequestAuthHandle$). If the session is not active, the $MetaTPMSessionManager'$ assumes that it was cached and therefore attempts to trigger the authorisation session resumption operation ($meta\_startTPM\_LoadAuthContext$ and $meta\_finishTPM\_LoadAuthContext$). After completing the interaction with the external session manager, it verifies whether or not the authorisation handle is indeed the one it is has requested; if not an error message is raised ($meat\_MetaTPMSessionManagerFailureCachedSessionWasDeleted$). In case the handles match, the channel $int\_readSessionParallel$ requests the state information of the session and $meta\_finishrequestAuthHandle$ communicates that data back to the caller.

$MetaTPMSessionManager''$ represents the counter operation of the latter process. It stores the state information of a certain authorisation session. If there is not enough free memory, it caches one of the active sessions and stores the state information in the memory block, which is vacant form now on.

$MetaTPMSessionManager'' =$
$meta\_startstoreAuthHandle?aH?nonce?keyHandle?secret \rightarrow$
$meta\_startsessioncount \rightarrow$
$meta\_finishsessioncount?spaces?authHandle1?authHandle2 \rightarrow$
if $authHandle1 = aH \vee authhandle2 = aH$
then $(meta\_writeSessionParallel!aH!aH!nonce!keyHandle!secret \rightarrow$
$meta\_finishstoreAuthHandle \rightarrow MetaTPMSessionManager'')$
else if $spaces = 0$
then $(meta\_writeSessionParallel2!1!aH!nonce!keyHandle!secret \rightarrow$
$meta\_increaseSessionCounter \rightarrow$
$meta\_finishstoreAuthHandle \rightarrow MetaTPMSessionManager'')$
else if $spaces = 1$ then
$(meta\_writeSessionParallel!N!aH!nonce!keyHandle!secret \rightarrow$
$meta\_increaseSessionCounter \rightarrow$
$meta\_finishstoreAuthHandle \rightarrow MetaTPMSessionManager'')$
else
$(meta\_startTPM\_SaveAuthContext!authHandle1 \rightarrow$
$meta\_finishTPM\_SaveAuthContext \rightarrow$
$meta\_writeSessionParallel2!1!aH!nonce!keyHandle!secret \rightarrow$
$meta\_increaseSessionCounter \rightarrow$
$meta\_finishstoreAuthHandle \rightarrow$
$MetaTPMSessionManager'')$

The initial event $meta\_startstoreAuthHandle$ receives the session state information from the caller, and then requests data about free memory blocks. If the session in question is already active, it reports the appropriate data back to the caller. On the other hand, if the session does not exist in the internal session storage, $MetaSessionStorage$ communicates that no or only one memory block is taken by another active session and stores the session state information in a vacant memory slot ($int\_writeSessionParallel2$). If no memory block is available, the caching mechanism will be triggered. Note that we do not elaborate on the caching strategy — hence, we always externalise the session that resides in memory block 1. The specification does not enforce a particular design. $meta\_startTPM\_SaveAuthContext$ and $meta\_finishTPM\_SaveAuthContext$ clear a memory block and save the information in the external session memory. After completion, $int\_writeSessionParallel2$ stores the session state in the vacant memory block.

9.1.5 The protected storage

The protected storage harbors all sealed objects. In our study we only need to store two objects. Thus, we designed the following CSP process:

$$ProtectedStorage = \\ ProtectedStorage'(O1, sec1) \;|||\; ProtectedStorage'(O2, sec2)$$

whereas:

$$ProtectedStorage'(id1, secret1) = \\ int\_loadObject!id1!secret1 \rightarrow ProtectedStorage'(id1, secret1)$$

The master process $ProtectedStorage$ initialises two sub-processes, each with a different object identifier (e.g. $O1$) and a corresponding authorisation secret (e.g. $sec1$). The two sub-processes are combined with the interleaving operator. Note that if we would attempt to design a precise model of the sealed storage, there would be interactions between different objects. However, whether or not the protected objects are stored inside a tree structure has no influence on this examination. The interactions between the stored objects and the owner of these objects are kept as simple as possible. Therefore we only provide a read operation ($int\_loadObject$). This read operation returns to the caller the authorisation secret that is required to successfully access the object.

9.1.6 The external session manager

The duties of the process $ExternalSessionManager$ are to handle the maintenance of cached sessions. It should receive session blobs and store them onto the hard disc drive, and, on request, retrieve the appropriate session blob so that $TPM\_LoadAuthContext$ is able to resume the session. However, since the specification does not enforce a certain design upon this process, it can be expected that the different implementations that will emerge on the market will vary from each other considerably. Moreover, [Pea02] states that this process does not have to be trustworthy. Thus, we not only implement a session blob storage and retrieval operation, but also include delete and inject session blob events that can be used by our intruder to interact with the external session manager. The overall CSP process consists of four sub-processes, each of which provide a different functionality.

All processes are interleaved with each other except $ExternalSessionManager'''$ and $ExternalSessionManager''''$. The $ExternalSessionManager''''$ synchronises on the channels $meta\_startTPM\_SaveAuthContext$ and $meta\_finishTPM\_SaveAuthContext$ to use the save authentication context functionality of $ExternalSessionManager'''$.

The $ExternalSessionManager'$ interacts only with the intruder. It offers the possibility of deleting a session blob.

$$\begin{aligned}&ExternalSessionManager' = \\ &\quad meta\_deleteSession?space \rightarrow \\ &\quad meta\_deleteSessionExtSessionStorage!space \rightarrow \\ &\quad ExternalSessionManager'\end{aligned}$$

$meta\_deleteSession$ takes the identification number of a stored session blob and deletes the session blob by calling $meta\_deleteSessionExtSessionStorage$ from the external session storage process.

The $ExternalSessionManager''$ communicates only with the intruder. It gives the intruder the opportunity to inject pre-designed session blobs.

$$\begin{aligned}&ExternalSessionManager'' = \\ &\quad meta\_injectSession?authHandle?nonce?digest?sec \rightarrow \\ &\quad meta\_storeSessionExtSessionStorage!N!(authHandle, nonce, digest, sec) \rightarrow \\ &\quad ExternalSessionManager''\end{aligned}$$

It accepts a session blob from the intruder (via $meta\_injectSession$) and stores this piece of data in the external session storage (via $meta\_storeSessionExtSessionStorage$).

The $ExternalSessionManager'''$ covers the operations that are necessary to cache an active authorisation session. This starts with creating the session blob and finishes with storing the blob inside the dedicated storage.

$$\begin{aligned}&ExternalSessionManager''' = \\ &\quad meta\_startTPM\_SaveAuthContext?authHandle1 \rightarrow \\ &\quad int\_startTPM\_SaveAuthContext!authHandle1 \rightarrow \\ &\quad int\_finishTPM\_SaveAuthContext?authContextBlob \rightarrow \\ &\quad meta\_storeSessionExtSessionStorage!N!authContextBlob \rightarrow \\ &\quad meta\_finishTPM\_SaveAuthContext \rightarrow ExternalSessionManager'''\end{aligned}$$

The channel $meta\_startTPM\_SaveAuthContext$ precedes the complete operation. The caller communicates which authorisation session should be suspended through this event. Since no one can interfere with the internal session memory, a verification whether or not the session in question is active can be omitted. The process forwards the authorisation handle to the $TPM\_SaveAuthContext$ process, via the event $int\_startTPM\_SaveAuthContext$. This process returns the appropriate session blob through channel $int\_finishTPM\_SaveAuthContext$. Finally, $meta\_storeSessionExtSessionStorage$ stores the session blob in the external session storage and $meta\_finishTPM\_SaveAuthContext$ signals to the caller that the suspend operation has been completed.

The $ExternalSessionManager''''$ represents the counter action of $ExternalSessionManager'''$. This process receives an authorisation handle and resumes the identified session by loading the appropriate data blob from the external storage.

$$
\begin{aligned}
&ExternalSessionManager'''' = \\
&\quad meta\_startTPM\_LoadAuthContext?authHandle \rightarrow \\
&\quad meta\_retriveSessionExtSessionStorage!authHandle?blob \rightarrow \\
&\quad meta\_startsessioncount \rightarrow \\
&\quad meta\_finishsessioncount?spaces?authHandle1?authHandle2 \rightarrow \\
&\quad \text{if } spaces = 0 \vee spaces = 1 \\
&\quad \text{then } int\_startTPM\_LoadAuthContext!blob \rightarrow \\
&\qquad int\_finishTPM\_LoadAuthContext?authHandle \rightarrow \\
&\qquad meta\_finishTPM\_LoadAuthContext \rightarrow ExternalSessionManager'''' \\
&\quad \text{else } meta\_startTPM\_SaveAuthContext!authHandle1 \rightarrow \\
&\qquad meta\_finishTPM\_SaveAuthContext \rightarrow \\
&\qquad int\_startTPM\_LoadAuthContext!blob \rightarrow \\
&\qquad int\_finishTPM\_LoadAuthContext?authHandle \rightarrow \\
&\qquad meta\_finishTPM\_LoadAuthContext \rightarrow ExternalSessionManager''''
\end{aligned}
$$

$meta\_startTPM\_LoadAuthContext$ receives the authorisation session identifier of the session that should be resumed. $authHandle$ is used to collect the associated session blob (via $meta\_retrieveSessionExtSessionStorage$). $meta\_startsessioncount$ and $meta\_finishsessioncount$ provide the process with information about whether or not session space is available inside the internal TPM memory. According to this information, $int\_startLoadAuthContext$ and $int\_finishLoadAuthContext$ activate the session by communicating with $TPM\_LoadAuthContext$. If no session space is available an active session is suspended. As in the process $MetaTPMSessionStorge$ we omit the implementation of a sophisticated caching strategy. Instead, $meta\_startTPM\_SaveAuthContext$ suspends the session that is stored in memory slot one. $meta\_finishTPM\_SaveAuthContext$ returns the appropriate session blob and channel $meta\_storeSessionExtSessionStorage$ stores that information in the external session storage. After resetting memory block one, $ExternalSessionManager''''$ picks up the activation process by communicating $int\_startLoadAuthContext$ and $int\_finishLoadAuthContext$. Lastly, the event $meta\_finishTPM\_LoadAuthContext$ informs the caller about the successful transaction.

9.1.7 The external session storage

The external session storage, in theory, provides space for an unlimited number of session blobs. Therefore, we parameterise the process with a list of quadruples.

$$
\begin{aligned}
&ExternalSessionStorage = \\
&\quad ExternalSessionStorage'(1, (N, N, N, N))
\end{aligned}
$$

$$
\begin{aligned}
&ExternalSessionStorage'(space, storage) = \\
&\quad meta\_retriveSessionExtSessionStorage!first(storage)!storage \rightarrow \\
&\quad SKIP \\
&\quad \Box \\
&\quad meta\_deleteSessionExtSessionStorage!space \rightarrow SKIP \\
&\quad \Box \\
&\quad meta\_storeSessionExtSessionStorage!first(storage)?newdata \rightarrow \\
&\quad ExternalSessionStorage(space, newdata) \;|||\; \\
&\quad ExternalSessionStorage(space + 1, (N, N, N, N))
\end{aligned}
$$

The process itself is designed to provide only the necessary operations for our model. The storage consists of one process per stored blob. The channel $meta\_retriveSessionExtSessionStorage$ synchronises on the values *authHandle* and on the session blob itself. The caller can use the *authHandle* to retrieve the desired data block. After the blob is delivered to the caller, the blob is erased. This is done by terminating the process. The delete operation works in a similar fashion. Finally, whenever a caller stores data inside the external session storage, a new $ExternalSessionStorage'$ process is spawned. Thus, there is always one more process in which to write.

9.1.8 The TPM owner

The process $Owner$ bears all the functionality necessary to launch and conduct an OS-AP session. In contrast to the TPM process, this process is much simpler because it does not need to perform internal operations. The process itself is divided into two parts.

$$
\begin{aligned}
&Owner(A, B, protocolcount, keyHandle, secret) = \\
&\quad meta\_startOSAP!A!protcount \rightarrow \\
&\quad meta\_getNonceNonceManager!A?nonceOddOSAP \rightarrow \\
&\quad output.A.B.Message1 \rightarrow \\
&\quad \text{if } protocolcount > 1 \\
&\quad \text{then } Owner'(A, B, keyHandle, secret, nonceOddOSAP) \\
&\quad \text{else } (Owner(A, B, protocolcount + 1, keyHandle, secret) \;|||\; \\
&\qquad Owner'(A, B, keyHandle, secret, nonceOddOSAP))
\end{aligned}
$$

$Owner$ is parameterised by the identities of the sender and receiver. In our case this is the label O for the owner and T representing the TPM. Further, it requires a numerical value (*protocolcount*) that keeps track of the number of launched authorisation sessions, an object identifier ($keyHandle$) and its corresponding authorisation secret (*secret*). The process starts with the signalling event $meta\_startOSAP$. This event indicates that an OS-AP session is about to begin; it also communicates the number of the session. $meta\_getNonceNonceManager$ requests a fresh nonce from the nonce manager. The next event (*output*) sends message 1 to the TPM. For further particulars about the OS-AP please see section 8.1. In theory the owner can initiate an unlimited amount of

sessions. To recreate this functionality, an $Owner$ process is interleaved by an $Owner'$ process. Thus, the overall process has the chance to proceed with the current protocol run or it can start a new one; of course it also has the option of launching the second protocol run at any given point in time. The process $Owner'$ handles the remainder of the OS-AP session.

$$
\begin{aligned}
&Owner'(A, B, keyHandle, secret, nonceOddOSAP) = \\
&\quad \Box\, authHandle : TauthHandle \bullet \\
&\quad \Box\, authLastNonceEven : NonceTPM \bullet \\
&\quad \Box\, nonceEvenOSAP : NonceTPM \bullet \\
&\quad input.B.A.Message2 \rightarrow \\
&\quad meta\_getNonceNonceManager!A?nonceOdd \rightarrow \\
&\quad meta\_calculateSharedSessionSecret?sessionsecret \rightarrow \\
&\quad output.A.B.Meassage3 \rightarrow \\
&\quad input.B.A.Message4 \rightarrow \\
&\quad SKIP
\end{aligned}
$$

$Owner'$ is initialised by the same parameters as the parent process and by the nonce ($nonceOddOSAP$) that is required to obfuscate the authorisation secret (in message 3). The rest of the process is self-explanatory. The owner receives message 2. Thereupon, it produces the components to compile message 3 (e.g. hash sum calculation) and sends the result to the TPM. Finally, it receives OS-AP message 4.

9.1.9 The intruder

The intruder process is based on the same framework as the intruder process in our OS-AP model. We had to extend the standard intruder model, since we not only consider the protocol, but also the internal activities that are triggered upon reception of protocol messages. Thus, the intruder should also have the power to take control over every non-trusted process. Note that the expression 'non-trusted' in this context refers to the TCPA framework, thereby describing a process that has no means of proving that it is working in a desired manner. In our model the external session manager represents such a process. The intruder can encourage this process to delete or inject any session data he wants. The

following CSP process models that behaviour:

$$
\begin{aligned}
&Intruder(IK, Storage) = \\
&\quad receive?O\,.\,T\,.\,x \rightarrow Intruder(IK \cup \{x\}, Storage) \\
&\quad \sqcap \\
&\quad \sqcap x : IK,\ O,T : Agent \bullet send\,.\,O\,.\,T\,.\,x \rightarrow Intruder(IK, Storage) \\
&\quad \sqcap\ deletesession \rightarrow Intruder(IK, Storage) \\
&\quad \sqcap \\
&\quad \sqcap (x, Y) : Deductions,\ Y \subseteq IK \wedge Y \subseteq Storage \bullet \\
&\qquad injectsession\,.\,(x, Y) \rightarrow Intruder(IK, Storage) \\
&\quad \sqcap \\
&\quad \sqcap (x, Y) : Deductions,\ Y \subseteq IK \bullet \\
&\qquad infer\,.\,(x, Y) \rightarrow Intruder(IK \cup \{x\}, Storage).
\end{aligned}
$$

The $Intruder$ process is initialised by the initial knowledge (IK) and a set containing all possible session blobs ($Storage$). The set IK contains the identifier of the other participants ($Owner$ and TPM), enough data to launch an OS-AP session (e.g. nonce), the identifier of the protected object that belongs to the intruder and its corresponding authorisation secret. The event *receive* is used to monitor the conversations between the owner and the TPM. After performing the *receive* operation, the intruder adds the collected message x to its knowledge. The process *intruder* can also send, via the *send* event, every message contained in its knowledge base. Moreover, as mentioned before, it can also delete (*deletesession*) or inject (*injectsession*) certain session blobs. Lastly, the intruder can make deductions upon its collected knowledge to gain new information. This *infer* operation is only available if the set Y is a subset or equal to its knowledge base. For more information see section 8 or [RSG+01].

9.1.10 The system

Figure 9.2 shows the structure of the overall system. In this picture we omitted the precise interactions between the different building blocks to increase its readability.

The intruder is connected to the external session manager process via *inject* and *delete* channels. The trusted platform itself is divided into an external (trusted sub-system) and internal area (TPM). The trusted sub-system contains the external session manager, the external session storage and the protected storage. The TPM includes processes to model the behaviour of OS-AP, the meta session manager, the internal session storage and the context management. The arrows indicate whether or not synchronisations between the various modules exist. After synchronising the processes we hide all channels except $meta\_TPMGrantAccessto$. This allows us to use a very efficient specification.

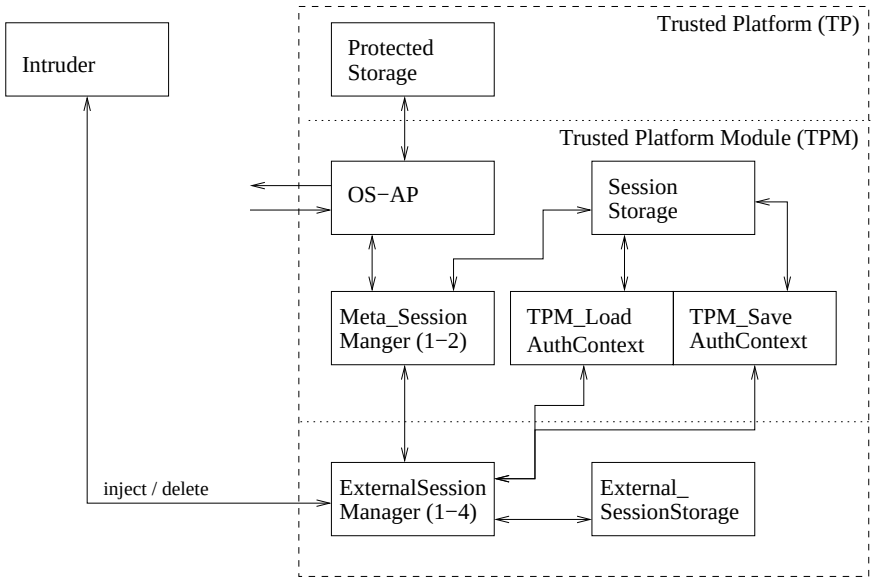

Figure 9.2: Session caching — the trusted platform

The specification is kept very simple since we have hidden nearly all events. The sole purpose of this specification is to determine whether or not the intruder can access an object ($O1$) that does not belong to him without knowing the corresponding authorisation secret ($sec1$). The following CSP process enables us to do so:

$$
\begin{aligned}
&Spec = \\
&\quad (\Box\, keyHandle : \{O1\}\bullet \Box\, y : \{O\}\bullet \\
&\quad meta_T PMGrantAccessto!keyHandle!y \rightarrow Spec) \\
&\Box \\
&\quad (\Box\, keyHandle : \{O2\}\bullet \Box\, y : \{I\}\bullet \\
&\quad meta_T PMGrantAccessto!keyHandle!y \rightarrow Spec)
\end{aligned}
$$

The process $Spec$ can only communicate the event $meta\_TPMGrantAccessto$ in a very restricted manner. Only the owner (label O) can access the object that is addressed by the key handle $O1$, and only the intruder (label I) can authorise to object $O2$.

9.2 The finite model

The model we have described so far is as (logically) accurate to the real world system as possible. In order to restrict the state space of the model, we apply the same techniques as in chapter 2.2 and 8. Therefore, at this point we will only briefly elaborate on our abstractions. They can be summarised in three points:

1. Instead of using the full OS-AP, we use the restricted version from section

8.1. This abstraction has far-reaching effects on nearly every data type involved (e.g. scope of session blobs).

2. We allow the owner to launch only 3 parallel sessions. This is enough to exercise the complete behaviour of the session caching system which, in turn, allows us to restrict the number of parallel sessions that are accepted by the TPM.

3. We combine certain processes to reduce (unintended) parallelism, via the external choice operator. For instance, we consequently combine the processes *TPM_LoadAuthContext* and *TPM_SaveAuthContext*. The resulting *AuthContext* process may not reflect the logical structure of the real world TPM as precisely as before, but this process structure is more accurate, from an operational point of view. In the real world both commands are atomic and the TPM seems to be single threaded (commands are always executed sequentially). Therefore, in the real world the TPM offers the choice between engaging in a *TPM_LoadAuthContext* or in a *TPM_SaveAuthContext* operation. Besides the *AuthContext* process, we internally merged *MetaTPMSessionManager* and *ExternalSessionManager* with their respective sub-processes.

9.3 Results

After applying all simplifications to our system we used FDR to evaluate whether or not the system refines our specification. FDR found various traces that violated our specification. One exemplary trace is appended in chapter A. In order to reduce the complexity of the traces, we modified the internal session memory so that it can only store information about one session at a time.

In the attack, the intruder pretends to be another TPM and starts a session with the TPM ($receive.T.T.(Msg1, Sq.\langle Na2, O2\rangle, \langle\rangle)$). The TPM acts as it should. It allocates the necessary resources by generating a new session identifier ($Ah1$) and loads the object $O2$ from the protected storage. Then it stores the authorisation secret of that object along with the nonce $Na2$ and the authorisation handle $Ah1$ in its internal session memory. Finally, it sends an appropriate reply ($send.T.T.(Msg2, Sq.\langle Ah1, Nt1\rangle, \langle\rangle)$) to the intruder. After receiving message two, the intruder starts another session with the TPM. This forces the TPM to clear the internal session memory. It uses the context management to store the state information of the first session in a session blob ($int\_finishTPM\_SaveAuthContextExternalSessionManager3.Storage.(Ah1, Na2, O2, sec2)$). Meanwhile, the owner of the platform tries to launch another OS-AP session ($send.O.T.(Msg1, Sq.\langle Na2, O1\rangle, \langle\rangle)$). The intruder intercepts that message and analyses its content. He learns the object identifier of the object that the owner tried to access ($O1$). With that identifier

he has all the information he needs to forge and to inject his own session blob into the external session storage ($meta\_injectSession.Storage.(Ah1, Na2, O1, sec2)$). To generate this faced blob, he uses the authorisation handle that identified his first session with the TPM ($Ah1$), a random nonce ($Na2$), the recently acquired object identifier ($O1$) and the authorisation secret to unlock *his* own object ($sec2$). Then he forces the TPM to resume the first (cached) session by sending message 3 of the OS-AP. The TPM receives the message, evaluates its internal session memory, discovers that the required state information is not available, and launches the context management to retrieve the information ($int\_startTPM\_LoadAuthContextExternalSessionManager4.Storage.(Ah1, Na2, O1, sec2)$). After loading the necessary information in the internal memory, the TPM compares the authorisation secret that was stored inside the session blob with the secret that could be deduced from message 3 (for further particulars see chapter 8). Since the intruder has injected his own authorisation secret, the two secrets match — and the TPM allows him to access the owner's object ($meta\_TPMGrantAccessto.O1.T$).

This attack is possible because the intruder can generate his own session blob on the one side and the TPM is unable to distinguish between faked and self-generated blobs on the other.

Interpretation As said before, the specification states [TCPA02] two particular properties about session blobs without stating how these attributes have to be achieved:

> *The contents of an authorisation context blob SHALL be discarded unless the contents have passed an integrity test.*

and

> *The contents of an authorisation context blob SHALL be discarded unless the contents have passed a session validity test.*

The first requirement deprives the intruder of the ability to alter session blobs and the second demand prevents the intruder from injecting his own faked session blobs. If the actual implementation of the session blob generation satisfies these two conditions, then the attack that FDR discovered clearly does not exist. It remains to be seen whether or not the first implementations really harbour possibilities that allow an intruder to successfully produce false session blobs.

This does not mean, however, that if we would add additional TPM functionality to our model, there would still be no other way to exploit the context management. A full-scale analysis of the TPM and the interactions between all features would be impossible for FDR. The only way to do so would be to show that certain features have no impact (directly or indirectly) on other features. This would give one the formal justification for pruning away non-related functionality. However, the development of the necessary compositional arguments with their formal ground-work lies beyond the scope of this thesis.

9.4 Abstracting low-level protocols

Within the trusted computing environment it is common that one has to verify hardware components that can interact with their environment, via certain protocols. The focus of such verifications lies not on the protocols themselves, but on the internal response of the addressed hardware component. Hence, it would be useful if we could find a way to reduce the model of external communication to only the necessary stimuli to exercise all possible behaviors (responses) of the hardware component. The internal transactions are usually complex enough to bring a complete state space exploration to its limits. In this section we will discuss certain properties of protocols such as authentication or secrecy and show how one can prune away the elements that guarantee these properties. This will result in a greatly reduced system which enables us to design the internal transactions of the sender or receiver in greater detail.

This work is an extension of the work done by Broadfoot and Lowe [BL03]. We will briefly review their work. Then we will analyse various properties of security protocols as well as correlate trace specifications that guarantee these properties.

We will split the integrity property into two categories: stream integrity and drop resistant stream integrity. Stream integrity demands that not a single message nor the communication stream itself can be altered. Drop resistant stream integrity on the other hand, requires that all messages that reach the destination are not modified during transit. Thus, messages can be lost without interrupting the protocol flow. We will distinguish between three properties, and one property will be divided into two subclasses. Hence, we will consider 6 different models.

1. authentication in combination with drop resistant stream integrity
2. authentication in combination with stream integrity
3. secrecy in combination with drop resistant stream integrity
4. secrecy in combination with stream integrity
5. authentication and secrecy in combination with drop resistant stream integrity
6. authentication and secrecy in combination with stream integrity (more details in [BL03])

As mentioned before this work is based on [BL03]. Broadfoot and Lowe discuss the general problem of verifying security architectures. Usually these protocols are divided in more than one layer. They give an example of a simple transaction protocol that deals with the purchase and payment of goods via an un-trusted network. The transaction protocol itself only provides the bare minimum of the

required security properties; e.g. it does not provide secrecy. This problem, in the today's industry, is solved by using another protocol such as secure socket layer (SSL [DA99]) as a foundation. This leads to a more complex overall protocol structure. If we use Casper and CSP to verify such a protocol this leads to an increased state space of the model.

Broadfoot and Lowe confront this problem by adjusting the standard threat model [LBH01] by pruning away the low level protocol (e.g. SSL). They do so in three steps:

1. they discuss the possible methods an intruder can perform in the protocol setting under observation;
2. from this Broadfoot and Lowe derive a model that satisfies the modified most general intruder description;
3. finally they show that their model still satisfies the required security properties.

Unfortunately [BL03] only includes the abstracted models of SSL and TLS; systems that provide authentication with stream-integrity and secrecy. We will take their approach and extend it so that it supports most of the scenarios that are likely to occur in the trusted computing environment.

9.4.1 Properties of protocols

Protocols can have various properties that ensure the faultless operation of a higher level protocol. It may be confusing to distinguish between the terms high and low level protocols. For these two expressions we will use the meaning that has been used within the crypto-protocol environment. Low level describes a protocol that can encapsulate another protocol (called the high level protocol), thus inheriting its security properties to the higher level protocol. Note this this is exactly the opposite of the definition within the computer programming environment.

[RSG+01] discusses various security properties such as: secrecy, authentication of origin, entity authentication, integrity, anonymity and various others. We will only focus on integrity, authentication (the classical definition) and secrecy. However we will discuss integrity in more detail, as mentioned before, we will divide the integrity property into two categories — drop resistant stream integrity and stream integrity.

Stream integrity represents the classical integrity requirement. No message within a given session, between two honest agents, can be altered, dropped or assume a different position then the intended one. The general integrity property is usually combined with the authentication property.

$$\forall a,\ b\ :\ Honest\,;\ s\ :\ Session\ \bullet\ tr \downarrow receives.b.a.s \leq tr \downarrow send.a.b.s. \tag{9.1}$$

This trace-specification expresses that within a given session s the receiver b can only process messages that honest agent a has sent to him. The symbol $\leq$ represents the prefix relation. Since the sequence of data items communicated on the receive channel are a prefix of the sequence communicated on the send channel it is impossible to drop messages between the start and the end of the protocol. Depending on whether the CSP model includes *start* and *close* signaling events, that are communicated on these channels, it may be that the last message can be dropped. Hence careful consideration should be given to such a phenomenon.

If the stream integrity property should be described without authentication the trace-specification looks slightly different:

$$\begin{aligned} \forall a,\ b\ :\ Honest\,;\ s\ :\ Session\,;\ i : Dishonest\ \bullet\ & \\ tr \downarrow receives.b.a.s \leq tr \downarrow send.a.b.s & \\ \vee & \\ tr \downarrow receives.b.a.s \leq tr \downarrow inject.i.b.s. & \end{aligned} \tag{9.2}$$

Similar to specification (9.1) this specification demands that the ordering of messages remains untouched. However this time the receiver cannot be certain whether the session was launched by the honest agent a or by the intruder.

Drop resistant stream integrity demands that messages cannot be altered without the knowledge of the receiver. In contrast to this, the classical definition of integrity requires that every message transmitted between the honest participants can not be modified as well as the communication stream itself. Hence, the protocol can 'lose' messages without violating the specification.

Applications for drop resistant stream integrity protocols can be real time protocols or secure video conference systems. Certain encryption standards do not require a complete data-stream to work properly. In general, real-time applications that are controlled via remote protocols can not afford to wait for retransmission of lost packets — as long as a certain proportion of packets reaches the receiver.

Another aspect may be that messages can assume different positions within the communication stream. Since we can not think of the usefulness of such a protocol we will omit the distinction between an enforced ordering and a lose ordering of messages. The trace specification for a protocol that allows the continuation after packet loss is as follows:

$$\begin{array}{l} \forall\ b\ :\ Honest\ ;\ s\ :\ Session\ ;\ \exists\ a : Agent \bullet \\ \quad tr \downarrow receives.b.a.s \preccurlyeq tr \downarrow send.a.b.s \\ \quad \vee \\ \quad tr \downarrow receives.b.a.s \preccurlyeq tr \downarrow inject.a.b.s \end{array} \tag{9.3}$$

$$\begin{array}{l} whereas\ : \\ \quad \forall\ tr' : Traces \bullet tr \preccurlyeq tr' \Rightarrow \\ \quad \exists\ f\ :\ dom\ tr \rightarrowtail dom\ tr' \bullet \\ \qquad \forall\ j \in dom\ tr \bullet tr(j) = tr'(f(j)) \\ \qquad \wedge \\ \qquad \forall\ x, y \in dom\ tr \bullet x \leq y \Rightarrow f(x) \leq f(y). \end{array}$$

The expression $tr \downarrow X$ hides the communications of all channels that are not within X, further it extracts the data values of the channels of X; leaving a sequence of data items. The function *dom* can be applied to a specific set of tuples[1] and returns the first element of each tuple.

This specification demands that the sequence of data elements that are communicated over the channel $receives.b.a.s$ is a subsequence ($\preccurlyeq$) of the channel $send.a.b.s$ or of the channel $inject.a.b.s$. The second half of the specification defines the subsequence operator ($\preccurlyeq$). In order to be a valid subsequence there has to exist a total injective function (called $f()$) that projects the domain of one trace to another trace. There are two restrictions upon this function. First, for all elements in the domain of trace tr (the elements the receiver processes) has to hold that the message that is on position j in trace tr is equal to the element $f(j)$ of trace tr'. This ensures the total injective property of function $f()$. This guards us from receiving messages that were not send by the originator. The second restriction demands that a message cannot outpace another message. Hence messages may be lost but can never swap position with other messages.

If the ordering is pushed to a higher level protocol it may very well be that out-of-order traffic is allowed as well. In this case the second part of the specification can be omitted. We will not further pursue this case in depth since, as mentioned before, we are not aware of a scenario where this is useful.

One has to be very careful with many layered protocols at this point: it may very well be that a certain state in a protocol run, that was *not* abstracted away, has to be reached in order to allow certain packets to be lost without resetting the current protocol session. So for instance could the protocol include an initial negotiation part. This negotiation could ensure that the sender and receiver agree upon certain conditions of the upcoming protocol run. Every lost message within

[1] A sequence can be defined as

$$seq\ X\ ==\ \{s : \mathbb{N} \rightarrow X \mid \exists\, n : \mathbb{N} \bullet dom\ s = 1..n\}$$

this provides us with tuples that consist of a position number and a data value.

these first messages would inevitably reset the protocol. However once agreed upon the conditions the sender and receiver switch into an operational mode that allows message loss without retransmission. An example for a $two - state$ protocol is the $TELNET$ protocol [PR83].

If the protocol has to reach a certain state before it allows message to be dropped, one has to include certain counters that guard this requirement. On the other hand it seems highly unlikely that a case arises were one can not collapse that functionality on to a higher level protocol and subsequently abstract that high level protocol away. Because of its rarity we will not further pursue this scenario.

Authentication or more precisely authentication of origin [RSG+01] demands that every message received by B within a session with A was indeed sent by agent A. Usually this also includes the integrity property. In our analysis we will adopt the same approach, the trace specification given in the integrity description (condition (9.1)) is valid. If you want to achieve proper authentication you have to include the identities of the participants in a direct or indirect manner. Hence if the intruder wants to breach authentication he has to alter the messages; this contradicts the integrity property. Therefore the integrity property may stand alone, however it is not possible for a protocol to enforce authentication but not integrity.

Secrecy is sometimes called confidentiality. There are two ways one can interpret this property. One can demand of the protocol that an intruder can not deduce elements that are defined as confidential. This interpretation of secrecy is easy to verify. The other version is more strict. It demands that the intruder can not deduce any information about the communication between the participants. This includes sending and receiving patterns in the communication stream as well as basic timing behaviour of all participants. For more information on the topic of information—flow see [RSG+01]. We will not further discuss this type of secrecy.

We can capture the simpler version of the confidentiality property by defining a set SEC that includes all messages (elements) that should be known only to the honest participants (set $Honest$) of a session. The event $signal$ provides us with the information about the legitimate users of the knowledge. The following trace specification captures the desired behaviour [LBH01]:

$$\begin{aligned} \forall\, messages\ :\ SEC\ \bullet\ signal.Claim\_Secret.a.b.messages\ in\ tr \\ \wedge\ a\ \in\ Honest\ \wedge\ b\ \in\ Honset\ \Rightarrow\ \neg(leak.message\ in\ tr). \end{aligned} \tag{9.4}$$

The specification expresses that all messages that are communicated on $signal.Claim\_Secret.a.b$, whereas a and b are honest agents, cannot be communicated on the $leak$ channel in the same trace. We only consider the case

where the complete data stream will be encrypted. Thus we consider every message sent in a protocol run between honest agents as secure. Using the definition above every message that is sent is also communicated on the *signal* channel. From this we can derive a intruder centered description of the specification:

$$\begin{aligned}&\forall\, i : Dishonest\,;\, a : Agent\,;\, s : Session\,;\, m : Message\,;\, tr' : Trace \bullet \\ &\quad (tr'^{\frown}\langle receive.a.i.s.m\rangle \leq tr) \Rightarrow \\ &\quad IIK \cup \{m \mid \exists\, a : Honest\,;\, i : Dishonest\,;\, s : Session \\ &\qquad\qquad \bullet\ send.a.i.s.m\ in\ tr'\} \vdash m.\end{aligned} \tag{9.5}$$

This trace specification guaranties that the intruder can only send a message to an honest participant if he has: received it from another agent, deduced it from his knowledge base or if it was in his initial knowledge (set IIK). (9.5) is only suitable for the case where secrecy and authentication are provided. In the absence of authentication we have to slightly adjust the specification (see condition (9.10)).

The standard intruder model according to Dolev-Yao [DY83] specifies various actions that can be performed by an intruder.

1. He can monitor messages that traverse the network.
2. He can use his knowledge-base to generate and send new messages.
3. He can drop messages that travel through his domain.
4. He can participate in a legitimate protocol session as an common agent.

We will not discuss this model in more detail since we already introduced the intruder process in section 8. Now we will focus in turn on every of the intruder's abilities and discuss how certain security goals will influence them. With the modified set of intruder capabilities we will design a new intruder model. Finally we will show that our model still satisfies the security properties *and* that our modifications have not restricted the intruder in an illicit manner.

9.4.2 Authentication only

To transform the standard model (as used in chapter 8) of a system in which the underlying communication between the participants of a session is authenticated, first we investigate the abilities that a intruder would have in such an environment.

1. *He can monitor messages that traverse the network.* This point remains unchanged since the protocol does not provide secrecy. Hence whenever the intruder in the old system learns a certain fact then the intruder in the new one does also.

2. *He can use his knowledge-base to generate and send new messages.* He can send messages with his own identity. However he cannot inject crafted messages into a session, except if he is one of the participants.

3. *He can drop messages that travel through his domain.* This may or may not be true depending on whether the protocol provides stream or drop resistant stream integrity with authentication. In the following model we will consider both cases. If attacker wants to drop messages he is able to do so if the underlying protocol only insists on drop resistant stream integrity. However he is never able to change the order of the messages.

4. *He can participate in a legitimate protocol session as a common agent.*

As mentioned before the standard intruder model places the intruder process between the sender and receiver process. For our low-level environment analyser we remove the intruder from this position and place him outside. See figure 9.3.

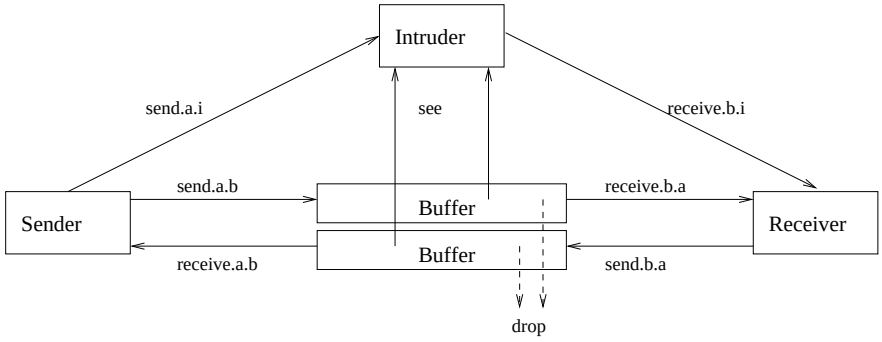

Figure 9.3: Authentication only

Since the intruder is still capable to withhold certain messages we place a buffer between participant A and B. As already discussed the intruder is still capable to overhear the traffic between A and B. To include this in our model we allow the intruder to inspect the content of the buffer. For reusability we recommend a three-way synchronisation on the $send.A.B$ event between the sender, the buffer and the intruder. The channels $send.a.i$ and $receive.b.i$ allow the intruder to communicate directly with the honest participants. If the protocol supports stream integrity, the following FIFO buffer is installed between the two channels:

$$
\begin{aligned}
&Buffer_{a,b} = |||_{s:Session} Buffer_{a,b,s}(\langle\rangle) \\
&whereas: \\
&\quad Buffer_{a,b,s}(sequence) = Buffer'_{a,b,s}(sequence) \triangleright Buffer''_{a,b,s} \\
&and \\
&\quad Buffer'_{a,b,s}(sequence) = \\
&\qquad send.a.b.s?m \rightarrow Buffer_{a,b,s}(sequence^\frown\langle m\rangle) \\
&\qquad \Box \\
&\qquad sequence \neq \langle\rangle \;\&\; receive.b.a.s.head(sequence) \rightarrow \\
&\qquad\qquad\qquad Buffer_{a,b,s}(tail(sequence)) \\
&and \\
&\quad Buffer''_{a,b,s} = send.a.b.s?m \rightarrow Buffer''_{a,b,s}.
\end{aligned}
$$

Every session possesses one buffer per communication direction. These buffers do not interfere with each other directly; hence they are combined by the interleaving operator. The process $Buffer'_{a,b,s(sequence)}$ requires the identities of the participants, the session identifier and a sequence of stored messages as input parameters. The buffer itself can receive (via the *send* event) or send (via the *receive* event) a message; the sequence that stores the messages is updated accordingly. The buffer does not contain additional channels that could be used by the intruder to inject messages. However the intruder can drop messages that traverse the channels. At this point we have to make the distinction between message and stream integrity. The buffer described above implements the stream integrity requirement. This means as soon as the buffer loses one packet in the data stream the complete session will be terminated since the buffer stops to forward packets. We used the sliding choice operator to accomplish an equivalent process.

The following CSP description shows the case where the protocol only provides drop resistant stream integrity:

$$
\begin{aligned}
&Buffer_{a,b} = |||_{s:Session} Buffer_{a,b,s}(\langle\rangle) \\
&whereas: \\
&Buffer_{a,b,s}(sequence) = \\
&\quad send.a.b.s?m \rightarrow \\
&\qquad (Buffer_{a,b,s(sequence^\frown\langle m\rangle)} \sqcap drop.a.b.s.m \rightarrow \\
&\qquad\qquad\qquad Buffer_{a,b,s}(sequence)) \\
&\quad \Box \\
&\quad sequence \neq \langle\rangle \;\&\; receive.b.a.s.head(sequence) \rightarrow \\
&\qquad\qquad\qquad Buffer_{a,b,s}(tail(sequence))
\end{aligned}
$$

The buffer has the same overall architecture. However a special *drop* event is included that indicates which message was lost (or intercepted) during transit. After the *drop* occurred the process returns to its initial state without flushing its temporary message storage (*sequence*).

Soundness of our model has to be discussed in two ways: first we have to show that our reduced model still satisfies the protocol requirements, second that we did not restrict the intruder in an illicit fashion.

To show that the communication stream of our model satisfies the authorisation trace specifications we have to determine the traces of every element within the overall system.

The buffer that is used in our stream integrity model can be described as[2]:

$$traces(Buffer_{a,b}(\langle\rangle)) = \{tr \in \{|receive.b.a, send.a.b|\}^* | \forall s : Session \bullet tr \upharpoonright receive.b.a.s \leq tr \upharpoonright send.a.b.s\}. \quad (9.6)$$

The buffer that satisfies drop resistant stream integrity can be described as:

$$\begin{array}{l} traces(Buffer_{a,b}(\langle\rangle)) = \{tr \in \{|\ receive.b.a,\ send.a.b,\ drop.a.b\ |\ \}^*\ | \\ \forall\ b\ :\ Honest\ ;\ s\ :\ Session\ ;\ \exists\ a : Agent \bullet \\ \quad tr \downarrow receives.b.a.s \preccurlyeq tr \downarrow send.a.b.s \\ \}. \end{array} \quad (9.7)$$

The intruder process is a derivation of the original intruder in the way it can communicate. The intruder can only communicate via the $receive.b.i$ event and can receive new information via the channels $send.a.i$ and $send.a.b$. He has no restriction on its knowledge base. This leads to following intruder:

$$\begin{array}{l} traces(Intruder) = \\ \quad \{tr \in \{|receive.a.i, send.a.i, send.a.b|a \in Honest \wedge b \in Honest|\}^*| \\ \quad\quad \forall b : Honest\ ; i : Dishonest\ ; s : Session\ ; m : Message\ ; tr' : Trace \bullet \\ \quad\quad tr'^\frown\langle receive.b.i.m\rangle \leq tr \Rightarrow IIK \cup collectedByIntruder(tr') \vdash m\} \\ whereas: \\ collectedByIntruder(tr') = sentToIntruder(tr') \cup \\ \qquad\qquad\qquad\qquad monitoredByIntruder(tr') \\ and \\ \quad sentToIntruder(tr') \hat{=} \{m|\ \exists a : Honest\ ; s : Session \bullet send.a.i.s.mintr'\} \\ and \\ \quad monitoredByIntruder(tr') \hat{=} \{m|\ \exists a : Honest\ ; b : Honest\ ; s : Session \bullet \\ \qquad\qquad\qquad\qquad send.a.b.s.mintr' \wedge a \neq b\}. \end{array} \quad (9.8)$$

Since we have to be general about the conduct of the protocol (about the communication flow) the only thing we can say about the traces of the honest participants is that:

$$\forall a : Honest \bullet tr \upharpoonright \{|send.a, receive.b|\} \in traces(Participant_a). \quad (9.9)$$

[2]The symbol $tr \upharpoonright X$ erases all elements in trace tr except those where channel X is involved and $\{|receive|\}$ represents the set of all messages on channel $receive$.

In other words if we reduce the traces of the overall system to the $send.a$ and $receive.a$ events, the resulting traces have to be traces that are within the traces of participant a.

Employing the semantic definition of the parallel composition[3] we can obtain the traces of the overall system by applying the restrictions of every participant to the initial traces ($\sum^*$). In other words every valid trace of the overall system has to satisfy the restrictions that every single participant harbors. This leads to following overall trace (for stream integrity):

$$
\begin{array}{l}
(For\ the\ Agents) \\
\forall\, a : Honest \bullet tr \upharpoonright \{|send.a, receive.b|\} \in traces(Participant_a) \\
\wedge \\
(For\ the\ Buffer) \\
\forall\, s : Session \bullet tr \upharpoonright receive.b.a.s \leq tr \upharpoonright send.a.b.s \\
\wedge \\
(For\ the\ Intruder) \\
\qquad \forall\, b : Honest\ ;\ i : Dishonest\ ;\ s : Session\ ;\ m : Message\ ;\ tr' : Trace \bullet \\
\qquad tr'^{\frown}\langle receive.b.i.m\rangle \leq tr \Rightarrow IIK \cup \\
\qquad\qquad\qquad\qquad collectedByIntruder(tr') \vdash m\}
\end{array}
$$

whereas :

$collectedByIntruder(tr') = sentToIntruder(tr') \cup monitoredByIntruder(tr')$

and

$sentToIntruder(tr') \hat{=} \{m |\, \exists\, a : Honest\ ;\ s : Session \bullet send.a.i.s.mintr'\}$

and

$monitoredByIntruder(tr') \hat{=} \{m |\, \exists\, a : Honest\ ;\ b : Honest\ ;\ s : Session \bullet$
$send.a.b.s.mintr' \wedge a \neq b\}.$

The requirement for authentication combined with stream integrity, as discusses above[4], are only met by traces that satisfy condition 9.1. We can observe that the trace specification of the buffer is already enough to restrict the traces so that they all satisfy the demanded authentication specification. All other trace restrictions only reduce the amount of legitimate traces even more, hence the overall system satisfies the condition (9.1).

The proof whether or not the system that handles authentication with drop resistant stream integrity satisfy condition (9.3) is equivalent to the latter one. It is already enough to observe the buffer to see that only, according to (9.3), legitimate traces are within the system.

The fact that our model does not restrict the system in an illicit manner can be seen directly. Since the traces of the buffer are precisely the same that are permitted by the authorisation trace specification, our buffer enforces only the

[3] $traces(P\ {}_A\|_B\ Q) = \{tr \in \upharpoonright A \in traces(P) \wedge tr \upharpoonright B \in traces(Q)\}$.

[4] $\forall\, a,\ b : Honest\ s : Session \bullet tr \downarrow receives.b.a.s \leq tr \downarrow send.a.b.s$

minimum restrictions upon the system. This applies to both cases, stream and drop resistant stream integrity.

9.4.3 Secrecy only

Having shown how one can obtain a model that provides us with an authenticated communication stream, we will now discuss, in similar fashion, how one can obtain a secure channel without authentication. As in section 9.4.2 we will discuss message and stream integrity separately from each other. If we assume that the protocol in use does not allow other agents to overhear important data, then the intruder has the following capabilities:

1. *He can monitor messages that traverse the network.* The intruder cannot obtain information about the content of the communication between honest participants. He can only add new information to his knowledge base whenever he directly receives a message or whenever he has obtained enough knowledge to deduce new facts.

2. *He can use his knowledge base to generate and send new messages.* Since the messages are not authenticated the intruder can inject messages into every session. This feature remains even if the overall system demands stream integrity. However in this case the intruder can not inject messages into an ongoing session: instead he has to launch the faked session between I_A and B himself. In this case agent A would not anymore be able to communicate with B within this session.

3. *He can drop messages that travel through his domain.* He can only do so if the protocol does not provide stream integrity.

4. *He can participate in a legitimate protocol session as an common agent.*

Similar to the model above (section 9.4.2) we construct a buffer that can be placed between the input and output channels of the honest participants. The intruder is transferred outside the direct communication channels (see Figure 9.4). Since we have no authentication the receiver of a message can not verify where the message originated from. Hence it should be possible for the intruder to inject certain packets into the buffer. The channel that provided the intruder in our last model with the information about the content of the communication has to be removed in order to satisfy the secrecy condition. Depending on whether we have stream or drop resistant stream authentication we include a *drop* event that allows the buffer to lose information. Figure 9.4 shows the general layout of the system. The intruder can inject messages, however only if he is the first that addresses the receiver.

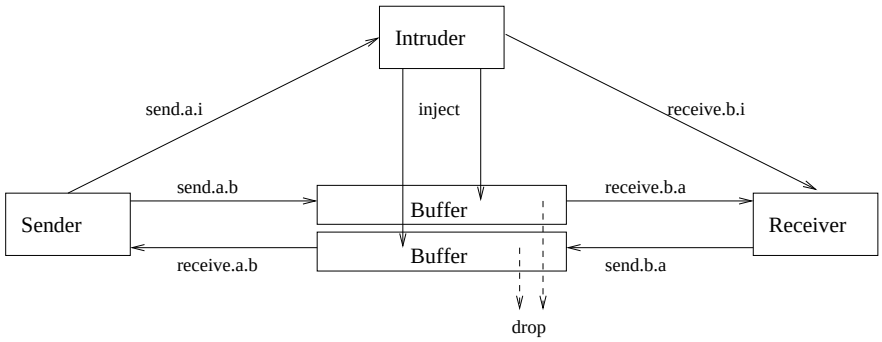

Figure 9.4: Secrecy only

First we will introduce the buffer that establishes connections that provide secrecy and stream integrity. The following CSP description satisfies our needs:

$$
\begin{aligned}
&Buffer_{a,b,i} = |||_{s:Session} Buffer_{a,b,i,s}(\langle\rangle, N)\\
&whereas:\\
&\quad Buffer_{a,b,i,s}(sequence, x) = Buffer'_{a,b,i,s}(sequence, x) \triangleright Buffer''_{a,b,i,s}\\
&and\\
&\quad Buffer'_{a,b,i,s}(sequence, x) =\\
&\qquad x \neq I \;\&\; send.a.b.s?m \rightarrow Buffer_{a,b,i,s}(sequence^\frown\langle m\rangle, A)\\
&\qquad \Box\\
&\qquad x \neq I \;\&\; inject.i.b.s?m \rightarrow Buffer_{a,b,i,s}(sequence, A)\\
&\qquad \Box\\
&\qquad x \neq A \;\&\; inject.i.b.s?m \rightarrow Buffer_{a,b,i,s}(sequence^\frown\langle m\rangle, I)\\
&\qquad \Box\\
&\qquad x \neq A \;\&\; send.a.b.s?m \rightarrow Buffer_{a,b,i,s}(sequence, I)\\
&\qquad \Box\\
&\qquad sequence \neq \langle\rangle \;\&\; receive.b.a.s.head(sequence) \rightarrow\\
&\qquad\qquad Buffer_{a,b,i,s}(tail(sequence))
\end{aligned}
$$

and

$$
\begin{aligned}
&Buffer''_{a,b,i,s} =\\
&\qquad send.a.b.s?m \rightarrow Buffer''_{a,b,i,s}\\
&\qquad \Box\\
&\qquad send.i.b.s?m \rightarrow Buffer''_{a,b,i,s}.
\end{aligned}
$$

At first sight the buffer seems more complicated then necessary. However since we guarantee stream integrity it is not possible for the intruder to inject or delete messages within an active session. Therefore once the agent a has sent one packet the intruder is not capable to inject packets anymore although we do not provide authentication. This switch is performed by introducing an additional tag, at each process, that determines what agent (A for honest agent and I for the

intruder) started the session. The only legitimate possibility for the intruder to send packets to agent b is via his direct channel ($receive.b.i$) or if he starts the session in the disguise of agent a. If the latter is the case agent a is not able to participate in the same protocol session. In case the buffer loses one message, the session is not continued; similar to section 9.4.2.

If we model a protocol that provides us with drop resistant stream integrity the buffer process becomes simpler:

$$
\begin{aligned}
&Buffer_{a,b,i} = |||_{s:Session} Buffer_{a,b,i,s}(\langle\rangle, N)\\
&whereas:\\
&Buffer_{a,b,i,s}(sequence, x) =\\
&\quad x \neq I \;\&\; send.a.b.s?m \rightarrow\\
&\quad (Buffer_{a,b,s}(sequence^\frown\langle m\rangle, A) \sqcap drop.a.b.m \rightarrow\\
&\qquad\qquad Buffer_{a,b,s}(sequence, A))\\
&\quad \Box\\
&\quad x \neq I \;\&\; inject.i.b.s?m \rightarrow Buffer_{a,b,i,s}(sequence, A)\\
&\quad \Box\\
&\quad x \neq A \;\&\; inject.i.b.s?m \rightarrow Buffer_{a,b,s}(sequence^\frown\langle m\rangle, I)\\
&\quad \Box\\
&\quad x \neq A \;\&\; send.a.b.s?m \rightarrow Buffer_{a,b,i,s}(sequence, I)\\
&\quad \Box\\
&\quad sequence \neq \langle\rangle \;\&\; receive.b.a.s.head(sequence) \rightarrow\\
&\qquad\qquad Buffer_{a,b,i,s}(tail(sequence)).
\end{aligned}
$$

This buffer is similar to the one above, except this one provides the option to drop packets within a session without resetting the current session.

Correctness Similar to the authentication part we have to show that our overall system still satisfies the trace-specification for secrecy with stream integrity and secrecy with drop resistant stream integrity. We will do so by first adapting the general secrecy specification to our needs, then we will investigate how the traces of our system relate to the specification.

The case where the abstracted protocol provides us with secrecy and drop resistant stream integrity demands that we alter our general secrecy specification (9.5). The traces of our new intruder contain one more channel that can be used to communicate with the environment. Our trace specification uses the output channels of the intruder to enforce a restriction upon the usage of available knowledge. Hence we have to include this new channel ($inject$). This results in following trace specification:

$$
\begin{aligned}
&\forall i : Dishonest\,;\, a : Agent\,;\, s : Session\,;\, m : Message\,;\, tr' : Trace \bullet\\
&\quad tr'^\frown\langle receive.b.i.s.m\rangle \leq tr \vee tr'^\frown\langle inject.b.i.s.m\rangle \leq tr \Rightarrow\\
&\quad IIK \cup \{m \mid \exists a : Honest\,;\, i : Dishonest\,;\, s : Session \bullet\\
&\qquad send.a.i.s.m \; in \; tr'\} \vdash m.
\end{aligned}
\tag{9.10}
$$

The intruder has slightly increased capabilities in a sense that he can inject messages in the buffers. The emphasis is on the restriction upon the messages the intruder can send via *inject* or *receive.b*. He can only send messages that where already in his knowledge, that some honest agent sent to him or messages that can be deduced from its knowledge base. The intruder that has to be employed in a system that provides us with stream or drop resistant stream integrity and secrecy can be described as follows:

$$
\begin{aligned}
&traces(Intruder) = \\
&\{tr \in \{|receive.b.i, send.a.i, inject.i.b | a, b \in Honest \wedge i \in Dishonest|\} * | \\
&\quad \forall b : Honest ; i : Dishonest ; s : Session ; m, m' : Message ; tr' : Trace \bullet \\
&\qquad (tr' \frown \langle receive.b.i.m \rangle \leq tr \Rightarrow \\
&\qquad IIK \cup \{m | \exists a : Honest ; s : Session \bullet send.a.i.s.m \ in \ tr'\} \vdash m\}) \\
&\qquad \wedge \\
&\qquad (tr' \frown \langle inject.b.i.m' \rangle \leq tr \Rightarrow \\
&\qquad IIK \cup \{m' | \exists a : Honest ; s : Session \bullet send.a.i.s.m' \ in \ tr'\} \vdash m'\})\}.
\end{aligned}
\tag{9.11}
$$

The overall system is generated by using the parallel composition operator $(P \ {}_A\|_B Q)$. Thus in order for a trace to be a legitimate trace of the overall system the trace has to satisfy the description of every participant.

The restrictions that the intruder enforces on its output channels are exactly those that are demanded to satisfy condition (9.10). In combination with the semantics of the parallel composition operator one can conclude that the system satisfies the secrecy specification. Moreover since the buffers are not included, this holds independently of the condition whether the underlying protocol supports stream or drop resistant stream integrity. Now we have to consider the stream integrity case. We have to show that our overall system satisfies the stream integrity specification. We do so by describing the traces of the buffer in use. The buffers traces can be described as follows:

$$
\begin{aligned}
&traces(Buffer_{a,b,i}(\langle\rangle)) = \{tr \in \{|receive.b.a, \ send.a.b, \ inject.i.b \ |\}^* | \\
&\forall a, \ b \ : \ Honest ; \ s \ : \ Session ; \ i : Dishonest \bullet \\
&\qquad tr \downarrow receives.b.a.s \ \leq \ tr \downarrow send.a.b.s \\
&\quad \vee \\
&\qquad tr \downarrow receives.b.a.s \ \leq \ tr \downarrow inject.i.b.s\}.
\end{aligned}
\tag{9.12}
$$

As in the cases before it is plain to see that the traces of the buffer are precisely those that satisfy the stream integrity property (condition (9.2)). Hence we can not only be certain that our overall model satisfies (condition (9.2)) but also that we did not over abstract.

If the protocol is resistant against message loss, we have to use our second buffer. This buffer has following traces:

$$traces(Buffer_{a,b,i}(\langle\rangle)) = \{tr \in \{|receive.b.a,\ send.a.b,\ inject.i.b\ |\}^*| \quad (9.13)$$
$$\forall a,\ b\ :\ Honest\ ;\ s\ :\ Session\ ;\ i : Dishonest \bullet$$
$$tr \downarrow receives.b.a.s \preccurlyeq tr \downarrow send.a.b.s$$
$$\vee$$
$$tr \downarrow receives.b.a.s \preccurlyeq tr \downarrow inject.i.b.s\}.$$

Again it is plain to see that our overall system satisfies the specification for drop resistant stream integrity. The traces of our buffer are the only traces allowed. This also means that we did not restrict our system more then necessary.

9.4.4 Authentication and secrecy

Since Broadfoot and Lowe [BL03] have already shown a way to simplify SSL and TLS (secrecy, authentication and stream integrity) we will only focus on drop resistant stream integrity. In such an environment the intruder has following capabilities:

1. *He can monitor messages that traverse the network.* Since the underlying protocol guaranties secrecy the intruder is not able to monitor the data stream between two honest participants.

2. *He can use his knowledge-base to generate and send new messages.* He may do so, however because of the authentication property he is not able to impersonate another honest participant. He can only communicate via channel *receive.b.i*.

3. *He can drop messages that travel through his domain.* He can always do so, however if the protocol enforces stream integrity the session will immediately stop (discussed in [BL03]). The case we will discuss below however, only demands drop resistant stream integrity, hence the intruder can drop any message he wants.

4. *He can participate in a legitimate protocol session as an common agent.*

Figure 9.5 shows the overall architecture from our system. It is very similar to the architecture from our authentication only system. The only difference — the intruder is not able to investigate the content of the buffer, due to the added secrecy property. This model employs the same buffer as in section 9.4.2 (condition (9.7)).

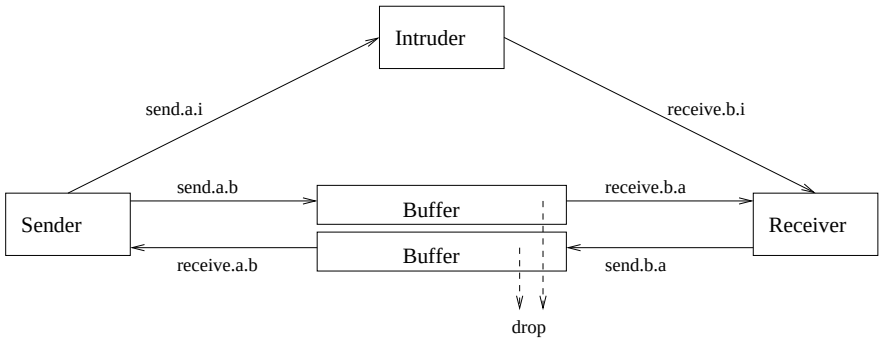

Figure 9.5: Authentication and secrecy

Soundness will be established in the same fashion as before. Since we have already the traces of the buffer (condition (9.7)) and participant (condition (9.9)) we only need to establish the set of traces for the intruder:

$$
\begin{array}{l}
traces(Intruder) = \\
\quad \{tr \in \{|receive.a.i, send.a.i|a \in Honest \wedge b \in Honest|\} * | \\
\qquad \forall\, b : Honest\,;\, i : Dishonest\,;\, s : Session\,;\, m : Message\,;\, tr' : Trace \bullet \\
\qquad tr'^{\frown}\langle receive.b.i.m\rangle \leq tr \Rightarrow IIK \cup sentToIntruder(tr') \vdash m\} \\
whereas : \\
\quad sentToIntruder(tr') \hat{=} \{m|\, \exists\, a : Honest\,;\, s : Session \bullet send.a.i.s.m in tr'\}.
\end{array}
\tag{9.14}
$$

This intruder is a combination of (9.8) and (9.11) in a sense that he lacks the *injection* channel and the facility to obtain information about the content of the secure protocol session. His knowledge base can only draw conclusion upon messages that were already in his knowledge (IIK) or that were directly send to him (via $send.a.i$).

Now we can directly establish the traces of the overall system. As mentioned before, the processes are tied together with the parallel composition operator. The traces of the system satisfy the following:

$$
\begin{array}{l}
(For\ the\ Agents) \\
\forall\, a : Honest \bullet tr \upharpoonright \{|send.a, receive.b|\} \in traces(Participant_a) \\
\wedge \\
(For\ the\ Buffer) \\
\forall\ b\ :\ Honest\,;\ s\ :\ Session\,;\ \exists\ a : Agent \bullet \\
\quad tr \downarrow receives.b.a.s \preccurlyeq tr \downarrow send.a.b.s \\
\wedge \\
(For\ the\ Intruder) \\
\qquad \forall\, b : Honest\,; i : Dishonest\,; s : Session\,; m : Message\,; tr' : Trace \bullet \\
\qquad tr' \frown \langle receive.b.i.m \rangle \leq tr \Rightarrow \\
\qquad IIK \cup \{m |\, \exists\, a : Honest\,; s : Session \bullet send.a.i.s.m in tr'\} \vdash m
\end{array}
$$

Since we have already established that the conjunct for the intruder matches the secrecy requirement (condition (9.10)) and the conjunct for the buffer is equivalent to condition (9.3) we can be certain that our model still satisfies the drop resistant integrity and authentication property. We can also see that this system does not satisfy the specification for stream integrity (condition (9.1)), since messages can be dropped. Furthermore we can be certain that we did not prune away to much detail. The traces of the intruder and the buffer perfectly match the secrecy, both integrity and the authorisation specification.

9.4.5 Conclusion

In this chapter we have shown how one can build a model of the TCPA's recommended session caching (and key caching) process. Our analysis showed that whenever one interprets the fuzzy parts of the specification in an unexpected manner vulnerabilities in the overall system arise. Since we used a specification as the foundation of our model, we can not claim to have found new weaknesses. One may only say that in case the specification is interpreted in the wrong way or certain parts are implemented erroneously errors will occur (similar to Chapter 3.5 and 3.4).

In our approach the TPM was not capable of determining whether or not a cached session data item was generated by the TPM. In this case the intruder could inject a session blob that allowed him to gain access to an object that belonged to another user without knowing the authorisation secret.

Our analysis revealed one problem with this type of verification process. Whenever we want to test the internal transactions of a hardware component and these internal activities depend on the input of a certain protocol or protocols, the state-space easily increases to an unmanageable level. Therefore the need arises for general abstraction techniques that on one hand prune away the functionality of the high level protocol that restricts the intruder in some way

(via its security properties) and on the other that do not restrict the intruder in its functionality.

To provide the security community with such a possibility we used Broadfoot and Lowe's paper [BL02] as a foundation and introduced a distinction between stream and drop resistant stream integrity. Drop resistant stream integrity requires that messages cannot be altered during transit. This however does not prevent an intruder form erasing certain data items form the data stream. Stream integrity on the other hand demands that every message in a particular session is received unmodified and no messages were dropped or injected. We discussed simplifications for six different protocol property combinations:

1. authentication in combination with drop resistant stream integrity
2. authentication in combination with stream integrity
3. secrecy in combination with drop resistant stream integrity
4. secrecy in combination with stream integrity
5. authentication and secrecy in combination with drop resistant stream integrity
6. authentication and secrecy in combination with stream integrity (more detailed in [BL03])

As mentioned before we did not include the case where the data stream within one session was not context sensitive. This leads to an enforced ordering of messages. We could not come up with a plausible scenario where it does not matter at what particular position a message is transmitted. On first glance this may be the case with various multimedia protocols that are capable of transferring great data files through the network. However as soon as the data reaches the application level the data has to be ordered. Hence even if the lower level protocol would permit an out-of-order communication the overall communication remains an in-order communication. For the unlikely event that one cannot transfer this enforced ordering to the higher level protocol[5] one has to change the FIFO buffer into a set. Other cases we did not consider are message integrity and secrecy or authentication without any notion of integrity. Message integrity allows an attacker to interfere freely with the communication stream, except that he cannot modify a message send within a session. We believe that these cases are seldom relevant within the trusted computing environment. If the need arises is should be straight forward to adapt our models accordingly.

The buffers we described are unbounded, hence not usable in a FDR supported analysis. For this, one has to restrict the buffer size appropriately. Broadfoot

[5]This may be the case if the hardware functionality that has to be investigated lies between the protocol layer and the layer where the data is ordered.

and Lowe have already discussed this issue in [BL03]. The buffer has to be of size N, whereas N is the maximum number of messages send into one direction without expecting a message back.

Finally we showed that our abstractions are sound in the sense that they satisfy the protocol requirements.

Chapter 10

Boot sequence

In this chapter we will present a CSP model of the boot sequence of a trusted platform. We will discuss the problems that may arise if we use the TCPA's integrity reporting systems for an integrity challenge-response protocol. The integrity challenge-response protocol is used whenever an external entity wants to determine whether a platform is trustworthy or not. In chapter 11 this entity will be a movie on demand service provider, hence for continuity reasons the example entity in this chapter will as well be a movie provider. This chapter closes with a discussion about the expressiveness of the TCPA's integrity metrics.

In TCPA terms there are three different ways to boot a trusted platform. The first choice is to boot the system without aid of the TPM and without verification. This leaves the system, after boot, in a non-trusted state. This corresponds to a standard boot as it is done in a usual PC. If the owner of a platform wants to use trusted agents or wants to operate with an external platform that demands a trusted system state, he can use two different ways to do so: the authenticated boot and the secure boot. The authenticated boot is the procedure we will look at more closely. Its main paradigm is that every security critical component has to be verified before it is executed and the result of the verification has to be stored in a tamper resistent memory. On the other hand, the secure boot takes this even further. After a component has verified the element that has to be executed next, it compares the result with reference values (stored inside the Data Integrity Registers). If these values deviate from each other the owner of the platform is informed.

Authenticated boot The precise procedure of an authenticated boot depends on the architecture upon which the trusted platform is based. The procedure we are using is the reduced boot cycle that is described in [Pea02]. This description focuses on the PC environment and mentions only the basic processes. For a full description see the PC specific TCPA specification [TCPA03g]. We believe no additional information can be extracted if we include the missing processes in our model. Figure 10.1 shows a sequence diagram of the authenticated boot cycle.

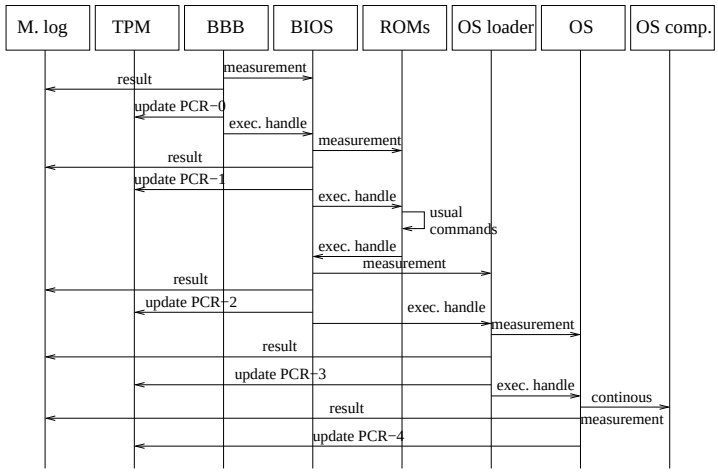

Figure 10.1: Authenticated Boot

The BIOS-Boot-Block starts the boot process. It verifies its own code, the integrity of the BIOS and records the results. The logging operation updates the PCR-0 and the trusted platform measurement storage (TPMS). As mentioned before the hash saved in PCR-0 ensures the integrity of the more descriptive measurement log file. Afterwards the BIOS-Boot-Block transfers the command to the BIOS. The BIOS performs its usual tasks, measures the Option ROMs, updates the measurement log and stores the digest in PCR-1. Then the execution handle is passed on to the Option-ROMs. They in turn execute their commands and, upon completion, hand over the execution right back to the BIOS. The BIOS verifies the OS-Loader, updates the trusted platform measurement storage and stores the measurement hash in PCR-2. After the OS-loader has received the execution handle, it performs its regular operations and measures the operating system (OS). The hash sum of that measurement result is stored in PCR-3 and the extended version is stored in the appropriate log file. Once the operating system has taken over, it evaluates the operating components as well as additional software. The paradigm here is that everything has to be evaluated before it is executed; of course, this applies only to code that changes the security state of the system. The OS stores its measured integrity digests inside PCR-4.

10.1 Model

We tried to implement a CSP model as efficiently as possible. In theory, the state space of the boot cycle is infinite. Only looking at the results for a single measurement makes this clear. In our model we reduce the nearly infinitely (2^{160} different results) many possible results to *ok*, *faulty* and *shut*. *ok* indicates that the integrity digest assumes a desired value. Hence the process that has been

evaluated behaves in the desired manner. Under desired behaviour in this case we understand that the application complies with the requirements of the integrity challenger (video on demand service provider). *faulty* on the other hand stands for the contrary — the requirements of the service provider are not met. This does not necessarily mean that the system is in an unstable or generally un-trusted state; another service provider may still find the integrity metrics sufficient. The *shut* value is a meta value introduced to signal that a particular application or process has not been executed. In the real world there is no such a value, instead the service provider has to check whether there is a banned application execution in the log files. The implementation of such a procedure would increase the state space dramatically and would gain no additional functionality for the integrity challenger or the dishonest owner of the platform.

Another abstraction that is necessary is the quantum of applications under observation. In the real world there is a myriad of security-relevant processes that can be started. However, we need only two applications to exercise all possible cases during the boot process and the preceding challenge-response phase. We name them *mediaplayer* and *pirate*. *mediaplayer* stands for the application that requests the right from the service provider to play a movie. The *integrity challenger* starts a challenge-response procedure to determine whether the system is running some illicit process that could copy the movie. In our model the illicit program is called *pirate*.

Furthermore, as mentioned before, we omitted certain elements of the boot cycle, hence we model only a reduced boot process.

Additionally, we made the assumption that the log file, stored in the trusted platform measurement store, is tamper resistant and accurate. This relieves us from the need to model the program control registers. In the real world the measurement log by itself is not tamper resistant. However since the program control register stores a hash sum of the log file no one is able to alter the log file unnoticed. By omitting the PCR we would allow the dishonest user to modify the registered system state. Such an abstraction would introduce many false positives. Hence, we design the trusted platform measurement storage as tamper resistant in the first place.

Finally, we prune away the message transfer of the challenge-response protocol. This abstraction is based on the assumption that our integrity challenger insists on vulnerability free protocols. Therefore, the dishonest user can never get information that was only designated for the *mediaplayer*. Note the procedure we are modelling cannot only be used for a video on demand system. The same or a very similar setting arises with any interaction that demands a rigorous integrity challenge-response between a trusted platform and another remote entity that wants to extend its chain of trust onto the platform. Examples for this are intrusion detection agents that want to verify whether the central management station can be trusted. Only if the station does not run illicit processes, agents perform a message exchange (e.g. share new signatures). We only focused on

the video on demand system since this chapter is considered to be one of the foundations for our digital rights management chapter (chapter 11).

System The overall system consists of 9 different processes. Figure 10.2 shows the interactions between the various processes. The arrows are named after the channels on which the processes at either end are synchronising. Moreover, the alignment of the arrows indicates the direction of the information flow. The processes are plotted by rectangles.

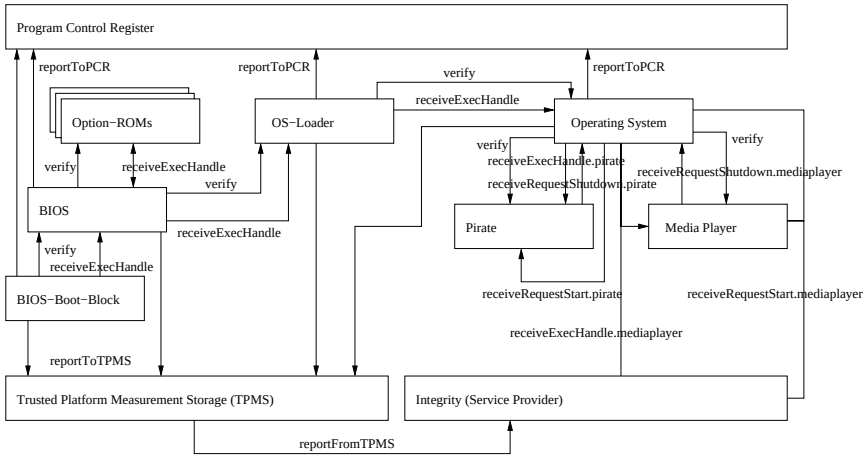

Figure 10.2: Boot sequence overall system

BIOS-Boot-Block The BIOS-Boot-Block is responsible for starting the construction of the chain of trust. It is the first code that is executed on the trusted platform. The CSP process that models the behaviour of the BIOS-Boot-Block is parametrised by its own name (*bbb*), the next entity in the chain of trust (*bios*) and the index of the program control register that must be used to store the integrity digest of the measurement result.

$$
\begin{aligned}
&BiosBootBlock(bbb, bios, pcr) = \\
&\quad startBootSequence \rightarrow verifyItself!bbb?value \rightarrow \\
&\quad reportToTPMS!bbb!value \rightarrow reportToPCR!pcr!value \rightarrow \\
&\quad verify!bios?value \rightarrow reportToTPMS!bios!value \rightarrow \\
&\quad reportToPCR!pcr!value \rightarrow receiveExecHandle!bios \rightarrow SKIP.
\end{aligned}
$$

The first event, $startBootSequence$, is a meta event that makes the resulting FDR traces more readable. Channel $verifyItself$ indicates that the process measures its own integrity[1]. $reportToTPMS$ and $reportToPCR$ are used to communicate

[1] One may wonder whether this measurement is really necessary, since in case the BIOS-Boot-Block was compromised it does not make sense to trust the measurement result. Hence one can only assume this piece of software is working properly.

the results of the self-evaluation to the trusted platform measurement storage and to the program control registers. Afterwards the same channels are used to verify the BIOS. Before the BIOS-Boot-Block terminates successfully, it transfers the execution handle to the BIOS (via $receiveExecHandle$).

BIOS The BIOS has basically the same structure as the previous process. It is parametrised by its own name ($bios$), the next two applications that have to be evaluated ($optionrom$ and $osloader$) and their corresponding program control register indexes (1 for the Option-ROMs and 2 for the OS-loader).

$$
\begin{aligned}
&Bios(bios, optionrom, osloader, pcr, pcr2) = \\
&\quad receiveExecHandle!bios \rightarrow executeOperations!bios \rightarrow \\
&\quad verify!optionrom?value \rightarrow reportToTPMS!optionrom!value \rightarrow \\
&\quad reportToPCR!pcr!value \rightarrow receiveExecHandle!optionrom \rightarrow \\
&\quad receiveExecHandle!optionrom \rightarrow verify!osloader?value \rightarrow \\
&\quad reportToTPMS!osloader!value \rightarrow reportToPCR!pcr2!value \rightarrow \\
&\quad receiveExecHandle!osloader \rightarrow SKIP.
\end{aligned}
$$

First, the BIOS receives the execution handle via $receiveExecHandle$ and executes its regular operations ($executeOperations$). Then it measures the option-ROMs, reports the measured result and submits the execution handle to the option-ROMs. After the option-ROMs have done their duty the execution handle is sent back to the BIOS. In the finial stage the BIOS verifies the OS-loader and transfers the execution handle over to the OS-loader after storing the measured values.

Option-ROM The option-ROM is actually of inferior significance. It is only parametrised by its own name ($optionrom$).

$$
\begin{aligned}
&OptionRom(optionrom) = \\
&\quad receiveExecHandle!optionrom \rightarrow executeOperations!optionrom \rightarrow \\
&\quad receiveExecHandle!optionrom \rightarrow SKIP.
\end{aligned}
$$

The process $OptionRom$ receives the execution handle, performs its usual tasks (e.g. resetting the card's start up values) and returns the execution token back to the BIOS.

Operating system loader The structure of the OS-loader CSP process is similar to the structure of the BIOS. It is parametrised by its own name ($osloader$), the next element in the chain of trust and the index of the PCR that can be used to store the integrity digest of the measurement result.

$$
\begin{aligned}
&OsLoader(osloader, os, pcr) = \\
&\quad receiveExecHandle!osloader \rightarrow executeOperations!osloader \rightarrow \\
&\quad verify!os?value \rightarrow reportToTPMS!os!value \rightarrow \\
&\quad reportToPCR!pcr!os \rightarrow receiveExecHandle!os \rightarrow SKIP.
\end{aligned}
$$

Initially the OS-loader receives the execution handle and performs the tasks for which it was designed. The channels $verify$, $reportToTPMS$ and $reportToPCR$ complete the measurement cycle of the operating system. Finally the right to execute its code is submitted to the operating system.

Operating system The operating system is divided in two processes, and both are parametrised by their own name and their corresponding PCR index. The process OS models the basic behaviour of the operating system that is necessary to reach a software state that enables the owner to execute other programs. OS' covers the regular operations of the operating system during its run-time.

$$
\begin{aligned}
&OS(os, pcr) = \\
&\quad receiveExecHandle!os \rightarrow executeOperations!os \rightarrow \\
&\quad OS'(os, pcr) \\
&whereas \\
&OS'(os, pcr) = \\
&\quad receiveRequestStart?agent \rightarrow verify!agent?value \rightarrow \\
&\quad reportToTPMS!agent!value \rightarrow reportToPCR!pcr!value \rightarrow \\
&\quad receiveExecHandle!agent \rightarrow OS'(os, pcr) \\
&\quad \Box \\
&\quad receiveRequestShutdown?agent?value \rightarrow \\
&\quad reportToTPMS!agent!value \rightarrow reportToPCR!pcr!value \rightarrow \\
&\quad OS'(os, pcr).
\end{aligned}
$$

The process OS receives the right to execute its code ($receiveExecHandle$), executes the set of initial operations ($executeOperations$) and moves into its second stage. The second stage symbolises the running operating system. Clearly in our model we only deal with, for our analysis, relevant activities. The process can either receive the request to start a new application (via $receiveRequestStart$) or close a running process ($receiveRequestShutdown$). On engaging in event $receiveRequestStart$ the operating system has to extend the chain of trust so that it also covers the software that has to be executed. It does so by using the same method as used in the previous processes. Finally, it allows the software agent, called a trusted agent, to have execution rights. Because more trusted agents can run concurrently, the process loops back on itself.

If the channel $receiveRequestShutdown$ is evoked, the operating system logs the shutdown in the trusted measurement storage and the program control reg-

ister. The *receiveRequestShutdown* event only becomes available through synchronisation if a trusted agent is willing to suspend its tasks exists.

Trusted agent The trusted agent sits on top of the operating system and is therefore the last element in the chain of trust. We are only including two different applications in our model. A media player (called *mediaplayer*) that can decrypt and play files from our service provider and a software that can tap between the decrypted data stream of the media player and the graphics adaptor (called *pirate*).

$$
\begin{aligned}
&Agent(name) = \\
&\quad receiveRequestStart!name \rightarrow receiveExecHandle!name \rightarrow \\
&\quad executeOperations!name \rightarrow receiveRequestShutdown!name!shut \rightarrow \\
&\quad Shutdown!name \rightarrow Agent(name).
\end{aligned}
$$

Every trusted agent follows a common behavioural template. First, the owner sends a request to the operating system to start a specific application (via channel *receiveRequestStart*). On receiving the resources that are needed to operate properly, the agent immediately starts to execute its operations. In the case of the media player this would be to decrypt the movie file and to display it. In case of the pirate software this would be to copy the movie data stream to a file on hard disk.

After they have completed their task, the owner can shut them down via the *receiveRequestShutdown* channel. This channel also instructs the operating system to act appropriately. The event *Shutdown* is not really necessary. It is an introduced meta event that should increase the readability of the FDR traces.

Trusted platform measurement storage This storage facility consists of 8 different processes that are interleaved with each other. Intuitively one may assume the best way to model a log file is by designing one process that harbours a sequence of tuples and every tuple consists of the name of the application and its status. This would force FDR to explore the behaviour of the storage process for every possible value of the sequence. This would dramatically increase our state space. Instead, we decided to apply a technique that is used to increase the efficiency of the standard intruder model [RSG+01]. We model every tuple that can be stored in the trusted platform measurement store as its own process. The process $TPMS'$ puts these interleaved processes together.

$$TPMS' = TPMS(bbb, shut) \;|||\; ... \;|||\; TPMS(pirate, shut).$$

Every storage cell (process $TPMS$) is parametrised by the name of the piece of code it stores information about and the result of the corresponding integrity

measurement. Initially every log entry has the value *shut*.

$$
\begin{aligned}
&TPMS(name, value) = \\
&\quad reportToTPMS!name?x \rightarrow TPMS(name, x) \\
&\quad \Box \\
&\quad reportFromTPMS!name!value \rightarrow TPMS(name, value).
\end{aligned}
$$

The memory block itself ($TPMS$) can only store a measurement result (via $reportToTPMS$) or communicate the content of its log entry (using $reportFromTPMS$). The channel $reportToTPMS$ is only synchronised with the appropriate trusted software to guarantee that no unauthorised entity can modify the content of the measurement storage.

Service provider The service provider process has a similar structure to the trusted platform measurement storage. We decided to model the integrity verification process in that way to increase the efficiency of the overall model (see $TPMS$). We tried to design the integrity evaluation to be as general as possible. For instance, it is possible for the service provider to cancel the process at any point.

$$
\begin{aligned}
&Integrity(x) = \\
&\quad receiveRequestStart!mediaplayer \rightarrow startIntegrityRequest \rightarrow \\
&\qquad (reportFromTPMS!x!ok \rightarrow Integrity'(bios) \\
&\qquad \sqcap errorIntegrityRequest \rightarrow Integrity(bbb)).
\end{aligned}
$$

The process $Integrity$ is parametrised by the first element (*bbb*) that has to be matched with the first log entry of the measurement log. The meta event $startIntegrityRequest$ indicates that a challenge-response is in progress. Afterwards the service provider has the choice to abort the integrity verification process ($errorIntegrityRequest$) or to accept the value of the pattern under observation. This evaluation is performed through a synchronisation of the $Integrity$ and the corresponding $TPMS$ process on the channel $reportFromTPMS$. If the trusted measurement log contains a value that differs from the demanded signature (in our case *ok*) then no agreement takes place. Thus, the challenge-response protocol cannot be finished and the media player never receives the permission to play the movie.

If the measurement log contains the desired value, the process $Integrity'$ is parametrised by the next element of the desired measurement log file (*bios*).

The process $Integrity'$ is similar to the previous process. Hence, we will not elaborate on the CSP code. If the trusted platform measurement storage contains the desired value for the *bios* entry, the challenge-response protocol moves into its next stage (*optionrom*). This kind of evaluation progresses until it reaches the last link in this integrity challenge-response process. The corresponding process

$Integrity^{VI}$ verifies whether any non-trusted application is running. It does so by inspecting the *pirate* log entry. The desired log entry must be *shut*; indicating that the application is not running.

$$
\begin{aligned}
&Integrity^{VI}(x) = \\
&\quad reportFromTPMS!pattern!shut \rightarrow \\
&\quad receiveExecHandle!mediaplayer \rightarrow Integrity(bbb) \\
&\quad \Box\ errorIntegrityRequest \rightarrow Integrity(bbb).
\end{aligned}
$$

The process $Integrity^{VI}$ uses the channel $reportFromTPMS$ to determine whether the pirate software is running or not (value *shut*). If this is true, the service provider signals the media player the right to play the movie ($receiveExecHandle$). In the real world this would be done by transferring the necessary cryptographic key (see chapter 11).

Specification The specification should determine whether it is possible for a dishonest user to run the media player and the pirate software at the same time. To enable us to design a specification as simple as possible we hide all events in the overall system except $receiveExecHandle$ and $Shutdown$. This is the bare minimum that is required to successfully keep track of the running trusted agents on the platform.

$$
\begin{aligned}
&Spec = \\
&\quad receiveExecHandle!pirate \rightarrow Shutdown!pirate \rightarrow Spec \\
&\quad \Box \\
&\quad receiveExecHandle!mediaplayer \rightarrow Shutdown!mediaplayer \rightarrow Spec.
\end{aligned}
$$

The specification offers the choice between starting the *pirate* or the *mediaplayer* process. Once an application is picked it, has to be closed first in order to allow the other application to be executed.

10.2 Results

We used FDR to verify whether our specification is refined by our overall system. FDR found various traces that all stem from the same problem. Therefore, we will only focus on one example to explain the underlying problem.

```
startBootSequence, verifyItself.bbb.ok, reportToTPMS.bbb.ok
reportToPCR.0.ok, verify.bios.ok, reportToTPMS.bios.ok
reportToPCR.0.ok, receiveExecHandle.bios, executeOperations.bios
verify.optionrom.ok, reportToTPMS.optionrom.ok, reportToPCR.1.ok
receiveExecHandle.optionrom, executeOperations.optionrom
receiveExecHandle.optionrom, verify.osloader.ok
```

```
reportToTPMS.osloader.ok, reportToPCR.2.ok
receiveExecHandle.osloader, executeOperations.osloader, verify.os.ok
reportToPCR.3.ok, receiveExecHandle.os, executeOperations.os
```

The first steps are as they should be: the chain of trust is established until the complete platform is operational (the operating system has taken over). This state is indicated by the event *executeOperations.os*. We omit a detailed description of every event, since the trace does not deviate from the desired behaviour that was described above.

```
receiveRequestStart.mediaplayer, startIntegrityRequest,
reportFromTPMS.bbb.ok, reportFromTPMS.bios.ok, reportToTPMS.os.ok
reportFromTPMS.optionrom.ok, reportFromTPMS.osloader.ok
reportFromTPMS.os.ok,verify.mediaplayer.ok
reportToTPMS.mediaplayer.ok, reportFromTPMS.mediaplayer.ok
reportToPCR.4.ok, reportFromTPMS.pirate.shut
receiveExecHandle.mediaplayer
```

The event *receiveRequestStart.mediaplayer* signals that the user wants to see the movie. For this, the operating system verifies the executable of the media player in the standard fashion. To receive the decryption key for the movie file the service provider verifies the system state via the dedicated channels (e.g. $reportFromTPMS.mediaplayer.ok$). Finally, the provider agrees upon $receiveExecHandle.mediaplayer$ and the file is decrypted and the movie is started.

```
receiveRequestStart.pirate, verify.pirate.ok
reportToTPMS.pirate.ok, reportToPCR.4.ok
receiveExecHandle.pirate
```

In the final stage of this attack the dishonest user simply starts the pirate software; thus both programs are running at the same time. This problem is often known as the *Time of check to time of use* problem. Clearly the vulnerability of the design lies in the absence of a continuous integrity challenge-response procedure. In the next section we will discuss the implications of this obvious attack.

10.3 Discussion

The result we obtained and the underlying problem have to be considered very carefully in order to come to the proper conclusions. Theoretically the chain of trust that is established during the boot cycle seems to be flawless in the sense that the *true* system state can always be determined. If the information about the system state were available at all times to the integrity challenger (service provider), then the system would be perfect.

A possibility for encountering the problem is to demand that a specific trusted agent, which allows other trusted agents to register their demands, is present at

any given point in time. This supervisory trusted agent continuously monitors the measurement log and the program control register. Whenever the system changes into a new state, a reevaluation for all registered agents has to take place. This strategy is similar to the *push model* of dynamic web services. Looking at TCPA specification 1.1b [TCPA02] the only way to enable the supervisor agent to retrieve the required state information is by evoking TCPA shielded capabilities. This requires the authorisation of the user. The only feasible way to handle this situation is to force the user to hand over his authorisation secret. With this secret, however, the supervisory agent could engage not only in the necessary commands but all functions. This is clearly unacceptable. Specification 1.2 [TCPA03d] remedies this drawback, by introducing a fine granulated delegation mechanism that allows a user to transmit execution rights of single functions to other entities.

Another issue that the TCPA leaves nearly un-addressed is to what degree, and to what extent or how the security relevant activities are measured. Focusing on the logging process of executed software (e.g. BIOS), Microsoft in its Next Generation Secure Computing Base (NGSCB) approach demands that the initial memory block that is reserved for the executable is logged and hashed. However they do not provide information on successive processes. During normal operation of the platform it may very well be that a user starts a trusted agent and this application needs additional library functions. The standard way to deal with this situation is that the application dynamically loads an additional library (in the Windows environment this is called a Dynamic-Link-Library (DLL)). That DLL contains the required functionality. The question that arises is whether or not the DLL code is verified before execution. It would be an easy task for a dishonest user to hide *additional functionality* in the library, thus acting in an illicit manner. If the memory block that is reserved for the DLL is measured as well, it will result in an altogether different problem. Clearly then the integrity measurement log would be very accurate. However, the log file itself would become extremely difficult to interpret.

Returning to the situation in our integrity challenger example: when we consider all the different BIOS versions with all their different configuration possibilities, it becomes clear that even considering merely the first link in the chain of trust a vast number of valid measurement results exist; adding other links of the chain of trust increases the number of legitimate log traces exponentially. The only feasible way to verify such a trace seems to be that the challenger uses the hash sums to verify the log file and then sends the different measurements to the vendor of the corresponding measured application. The vendor could then determine whether or not the settings and the executed code are compliant with the security specifications. This would somewhat tackle the problem of evaluating a simple boot cycle without the above mentioned dynamic link library problem.

Let us assume that we have a system that measures every executable (including dynamic link libraries) and that there exists an infrastructure that allows

the service provider to *out-source* the evaluation of the measurement log to the vendors of the corresponding applications. Now considering the case where a user uses his business PC and an office suite during the day and, after work, he wants to launch an trusted application (watch a movie on demand) that requires an integrity-challenge response procedure. The fact that the measurement log carries the complete history of the working day tremendously increases the resources required to evaluate one application. In addition to that fact, some users may decide just to log off their machines instead of shutting them down. This would mean that every call from a specific application for an additional executable over the last few days would have to be verified.

This example shows that such an approach does not scale well with the run time of the trusted platform (e.g. DLLs).

Finally, the question on how one can determine that a specific software is not running is another un-addressed issue. In terms of our example, the service provider has to possess a reference list that contains every forbidden application, including every version of each application. Even if he has such a file, a sophisticated dishonest user could write his own illicit application, which would then clearly not be on the reference list.

It remains interesting to see how the TCPA tries to solve these problems.

10.4 Conclusion

In this chapter we have investigated the first steps of the chain of trust that are vital for the trusted computing environment. We have designed a CSP model involving the BIOS-Boot-Block, the BIOS, option-ROMs, OS-Loader, the operating system and various applications. To test whether the collected information about the system state can be used to perform a proper integrity challenge-response, we added a service provider that would allow an installed application (*mediaplayer*) to execute only if the platform was in a desired state.

FDR provided us with a very obvious example that defied the demands of the service provider. The underlying flaw is rooted in the fact that no service is provided that continuously monitors the system state and that matches a changed setting against temporal security requirements (e.g. *pirate* is not running as long as *mediaplayer* is executed).

The chapter was closed by a discussion about the implications of introducing a supervisory application and about the expressiveness of the measurement log itself. The supervisory agent has to keep track of state transition within the trusted platform and if necessary, re-initiates an integrity challenge-response procedure to determine whether some application specific security policies are violated.

A final note concerning the value of the FDR counter-example: we do not know whether this issue has not been addressed because of marketing-political

reasons or because the specifying body did not investigate carefully enough the specification on an operational rather than on a functional level. Hence, we cannot claim that our model reveals a novel attack. However, what we can say at least is that our model and the underlying technique enabled us to think thoroughly about the overall system.

Chapter 11

Digital Rights Management

In this chapter we will look at one application that could profit from the enhanced feature set of a TCPA trusted platform. There are various interesting environments that profit from the functionality we have described so far. We chose Digital Rights Management (DRM). DRM is the management, maintenance and enforcement of accessibility rules upon digital content. Within that area we focus on a simple video on demand infrastructure. The goal of this chapter is to give an example that there are scenarios where the attacks discovered in 8 lead to serious vulnerabilities.

We will suggest a DRM protocol that should ensure that the user can download a movie and that he can only use the obtained file in a legitimate fashion. In this case the service provider determines the legitimacy of an action. We will use the AACP (Asymmetric Authorisation Change Protocol) to design a protocol that suites our purpose.

DRM is a frequently discussed tropic in today's media, especially since Microsoft and the TCPA presented architectures that would allow the enforcement of reliable rights management [And03, Pfi02, Plu02b, Plu02, Him02, Pfi02]. The topic itself is very broad and most of the main issues within that area have a wide scope. Therefore, before we start describing our approach, we will give an overview of DRM.

11.1 Review of current problems and solutions

DRM systems are becoming increasingly important in a world that relies more and more on the digital distribution of services (i.e. content). As with non-digital transactions, the seller or producer of goods does not want a customer to reproduce the goods without his permission. Hence, the right owner of the protected content must have continuous control over the content even when it leaves the owner's digital sales table. More precisely, DRM strives to guarantee the following:

1. It should not be possible to consume content without the permission of the intellectual-property owner.

2. It should protect the content from being modified in an illicit manner.

3. It should identify protected content with the intellectual property owner.

4. It should prevent unintended reproduction of content.

Protection mechanisms that safeguard such restrictions, can be implemented in various ways. The most prominent way of distinguishing between the various approaches is by implementation level. For example, *iTunes* resides in the application layer [Ver01]; Microsoft's Rights Management System (RMS) originates in the operating system level [Mic04]; and the Content Scramble System (CSS) was an attempt to root the DRM protection on the hardware level [Kes00].

Another way to classify DRM systems is by their architecture. [Par00] introduces a common taxonomy [AH04]. [Par00] distinguishes between systems that use or do not use a virtual machine, that use fixed embedded or external rights sets and that use a direct or indirect communication line between the service provider and the customer.

The virtual machine is a trusted agent that is installed on the customer's platform and has to be obtained from the service provider. The digital content can only be accessed via this trusted agent. The DRM model we are discussing makes use of such a virtual machine. The rights set contains all necessary information about the operations the customer is allowed to perform. These sets can be fixed sets that are defined once, and, thereafter, are applied to all DRM protected content. These sets are usually incorporated in the virtual machine. Another option for defining the control sets, is be to embed the rights sets in the content. This would make the whole DRM system more flexible because they can easily vary from content to content. Finally, the sets could be stored externally. Therefore, whenever the customer wants to use the DRM protected data, the virtual machine must request the appropriate set to determine whether the requested operation is permitted.

The protocol we discuss uses a virtual machine (trusted agent) and an external rights set.

Mainly negative points of view are taken in today's news [mic03, Cli04]. This is due to some serious issues that have not yet been fully addressed. The main issues are:

1. Privacy - in order to keep track of the digital content, identification numbers have to be introduced. This allows service providers to keep track of certain behaviour of the user.

2. Fair use - the user's legal rights sometimes clash with the rights granted by the service provider. So, for instance, one is allowed to burn backup

copy of a music Compact Disc (CD). DRM systems with a coarse granular rights set may not allow this. More generally, the DRM may not permit all actions that are legally permitted [AH04].

3. External control - the DRM system gives the service provider certain control over the user's platform [And03].

This introduction is far from complete and the interested reader is referred to [AH04, RD03, Lyo01, Par00] for an introduction in DRM. For a more elaborate requirement analysis on DRM participants see [BCP+99]. [MHB03] discusses the interference of DRM systems with the legal rights the purchaser of digital content can expect. Additional information about the legal rights and implications of over restriction through DRM systems can be found in [Gro03, WIPO04].

11.2 The integrity challenge-response protocol

Clearly we cannot include all problems that arise because of legitimate transfer and modification of rights. We will omit these issues in the following design. Thus the only resulting requirement for our protocols is:

> The owner of the platform must not use the digital content without the direct permission of the rights owner.

The basic architecture for our approach consists of a service provider, a user that may or may not be honest and a fully certified trusted platform.

In section 10 we introduced a term called integrity challenge-response protocol. For our boot sequence model it was not important to consider the integrity challenge's precise transactions between the remote entity and the trusted platform. For the following discussions however we have to take a closer look at a version of this protocol. The following protocol intends to allow a remote entity to store data securely on a trusted platform. Secure in this sense means that the data is only available if the system is in a trusted system state. We will quote directly from [Pea02]:

> Another choice is to build encrypt the data at the target. This uses a variant of the standard challenge/response protocol, called by TCPA an "integrity challenge". The source sends a nonce to the target platform, which signs the nonce with a TP identity key and incorporates the current PCR values into the signature. The target returns to the source the signed nonce and PCR values. The source verifies that the nonce is correct, verifies that the signature was done using an identity key of a genuine TP, and verifies that the PCR values indicate a safe software environment. The source sends plain test data to the target. The target then uses the bulk encryption to encrypt the plain text

data and wraps the bulk encryption using a non-migratable wrapping key bound to certain PCR values.

At this point we omit a *Casper* driven analysis because it is manifestly apparent that this protocol does not work. If we use our standard intruder scenario, the intruder could overhear the plain text. Since the plain text is not protected, he could circumvent the on DRM based restrictions. The assumption that the challenger uses an encryption protocol as a basis for the suggested protocol eliminates this attack possibility. Another issue is that this approach does not properly authenticate the platforms to each other. This leads to another obvious attack scenario.

The dishonest user could have two platforms: one in an un-trusted condition and another in a trusted software state. He could use the un-trusted platform to trigger the action that requires the integrity challenge response. The integrity challenger would then establish an encrypted (e.g. SSL) connection with the platform and send his nonce (Na)[1]. Afterwards, the dishonest platform launches an integrity challenge with the trusted computer and uses the nonce Na. The trusted platform answers in the obvious way and returns the singed nonce and the PCR values. The dishonest computer then injects the received data into the SSL connection. The initial integrity challenger then finishes the protocol run by sending the plain text data. Again, the dishonest user could use the data at his will. This problem could be solved if the trusted platform identity key that is used to sign the integrity response could be linked to the identity that is used for the SSL based authentication. Unfortunately, trusted platforms possess an identity key system (see privacy certification authorities (P-CAs) Chapter 7) that should prevent an entity from correlating information about the activities of a platform. Therefore this solution would not bode well with the demands of the privacy community.

Nonetheless, at this point we assume that the integrity-challenge is built upon a protocol that ensures authentication, integrity and secrecy Moreover, the authentication is directly or indirectly tied to the identity key that has been used to sign the nonce and the PCR values. Indirectly in this sense means that a trusted third party vouches that the SSL identity and the identity key refer to the same TPM. This solution could solve the privacy concerns.

An altogether different solution is to include the challenger's and responder's IDs in the integrity response. For our AACP based DRM protocol we will only use the first part of this procedure (we will omit the transmission of the plain text and its successors).

Assuming now that this procedure is doing what it is supposed to do, little more is required to complete a DRM protocol.

[1]Note that the SSL connection guarantees authentication and secrecy. However, this authentication refers to a host-to-host authentication rather than an identity-to-identity authentication.

11.3 The AACP version

The approach's purpose is to give an example of that even if the attack upon the AACP (see chapter 8) found earlier had no immediate impact, it still can have one if another protocol wants to use the protocol as basis for a more complex protocol. First we will describe the general procedure of the DRM protocol and then we will extend the AAC protocol. Finally, we will discuss the vulnerabilities of the system.

The key part of the protocol is the creation of an encrypted object that contains the symmetric key for the encrypted data file. This key object is created via the ADIP, and the authorisation secret is changed via the AACP to a value that is unknown to the user. This key object is stored on the user's side and the corresponding object handle is stored inside the code of the trusted agent. Initially the trusted agent also contains a public/private key pair that is linked to his ID and a hash sum of the private key. Once the AACP session is completed, the trusted agent requests the authorisation secret to initialise the key object with his private key. After he has stored his private key inside the object he erases the private key from his code base. This should prevent a sophisticated user from reverse engineering the key (i.e. extracting it form the binaries).

Whenever the user requests a movie file it is symmetrically encrypted and the symmetric key is encrypted with the public key of the trusted agent. If the user wants to access the restricted data, he has to launch the trusted agent. The trusted agent requires the authorisation secret of the key object to read his private key and to subsequently unlock the data file. Thus, he requests the secret from the service provider. This communication is encrypted via SSL. The service provider performs an integrity challenge and verifies whether or not the system state is agreeable and whether or not the user has sufficient rights to access the data. If so, he transmits the authorisation secret.

At this point we assume that the vulnerability discussed in chapter 10 is solved and that the integrity challenge response procedure is also without vulnerabilities (no SSL session hijacking). Therefore, the service provider can be sure that the system is in the desired state and that the only entity receiving the authorisation secret is the trusted agent. The trusted agent uses the received information to access the decryption key for the file. More precisely, the trusted agent unseals the key object, uses the hash sum to verify the obtained private key and finally decrypts the symmetric key. Note that the key object is not only tied to a specific authorisation value, but also to a specific TPM. Thus, it is impossible for another TPM to access the object. Furthermore, the key object can only be accessed if the systems PCR has the right value. This approach has the following advantages (mostly for the service provider):

1. It is more flexible. The service provider can change rights basically on the fly. Clearly this advantage has terrible disadvantages for the user. [And03]

discusses the scenario where the service provider can remove the rights of any user at good will in great detail. For our extremely simplified DRM this is, however, of no concern.

2. It provides more security against sophisticated attacks, since the authorisation secret / private key is stored outside the trusted agent's executable. Another solution would be to encode it into or store it in the trusted agent. This, however, would allow a dishonest user to reverse-engineer certain parts of the trusted agent, and obtain the authorisation secret through this.

11.3.1 Analysis of the protocol

We will only focus on the core of our small DRM protocol - the change of the authorisation secret of the key object. For this we extend the AACP of section 8 to a three participant system. Figure 11.1 shows the information flow between the owner of the platform, the TPM and the service provider.

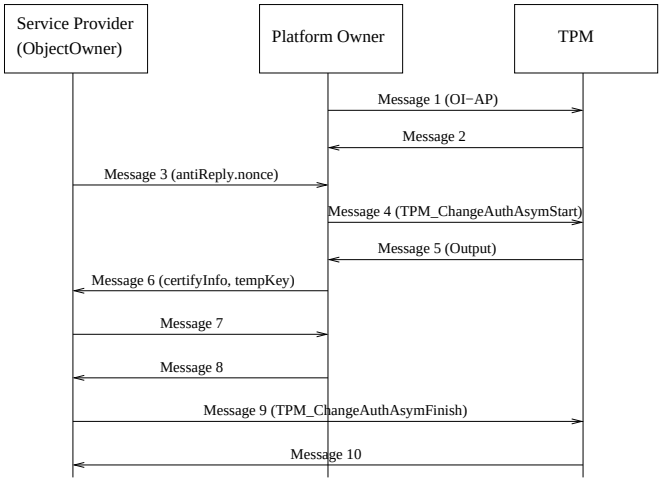

Figure 11.1: The extended AACP

Since we already described the AACP, we will only focus on the parts that have to be changed. The general succession of commands is changed slightly [Pea02]. The platform owner initiates the OI-AP session, submits the command *TPM_ChangeAuthAsymStart* and receives the output. The second command however (*TPM_ChanceAuthAsymFinish*) is sent by the object owner [Pea02]. Since the command *TPM_ChangeAuthAsymStart*, sent in message 3 of the original protocol includes the data value *antiReply.nonce* and this value should be provided by the object owner, we include a message that sends that data value

from the object owner to the platform owner (*Message 3*).

Message 3.	$ObjectOwner \rightarrow Owner$	:	$antiReply.nonce$
Message 4.	$Owner \rightarrow TPM$	:	$tag, paramSize, ordinal, idHandle$ $antiReply.nonce, tempKey$

The specification mentions this transfer only indirectly. Thus, we do not include additional data or manipulate the data in any way (i.e. encrypt). The TPM then acts as described in chapter 8. Once the platform owner has received the output of the command, he has to communicate certain information to the object owner. We will quote this part from the specification:

> It is envisaged that *tempkey* and *certifyInfo* are given to the owner of the entity whose authorisation is to be changed. That owner uses certifyInfo and a TPM_IDENTITY_CREDENTIAL to verify that tempkey was generated by a genuine TPM.

We introduce the missing protocol communication (*Message 6*) only consisting of *tempkey* and *certifyInfo*:

Message 5.	$TPM \rightarrow Owner$	:	$tag, paramSize, returnCode, certifyInfo$ $sigSize, sig, ephHandle, tempKey$ $nonceEven, continueAuthSession$ $HMAC(IDKey.usageAuth, data)$
Message 6.	$Owner \rightarrow ObjectOwner$	:	$certifyInfo, tempKey$

The object owner verifies the received data by using various public certificates and prepares to submit the command *TPM_ChanceAuthAsymFinish*. Various parameters are required for this command :

Message 9.	$ObjectOwner \rightarrow TPM$	:	$tag, paramSize, ordinal, parentHandle$ $ephHandle, entityType, newAuthLink$ $newAuthSize, encNewAuth, encDataSize$ $encData, authHandle.Parent, nonceOdd$ $continueAuthSession$ $HMAC(parentKey.usageAuth, data)$

We will omit a description of the meaning of every parameter since we have already described them in chapter 8. It is only important to notice that to guarantee the integrity of all parameters, the authorisation secret of the parent object is required (*HMAC(parentKey.usageAuth, data)*). However, the object owner is not in possession of that secret, and the platform owner cannot transmit the secret. If he would transmit the secret otherwise, the object owner could unlock more than just one protected object. This leaves only one choice —

the object owner generates the values *entityType, newAuthLink, newAuthSize, encNewAuth, encDataSize, encData* and *nonceOdd* and transmits them to the owner of the platform (*Message 7*).

Message 7. $ObjectOwner \rightarrow Owner$: $tag, paramSize, ordinal, parentHandle$
$ephHandle, entityType, newAuthLink$
$newAuthSize, encNewAuth, encDataSize$
$encData, authHandle.Parent, nonceOdd$
$continueAuthSession$

The owner of the platform then generates the $HMAC$ over all parameters and returns the resulting digest.

Message 8. $Owner \rightarrow ObjectOwner$: $HMAC(parentKey.usageAuth, data)$

Then the object owner submits (see *Message 8*) the full command to the TPM. The TPM processes the command and returns the output to the object owner (*Message 10*).

Message 10. $TPM \rightarrow ObjectOwner$: $tag, paramSize, returnCode, outDataSize$
$outData, saltNonce, changeProof$
$nonceEven, continueAuthSession$
$HMAC(parentKey.usageAuth, data)$

For the further course of this analysis it is important to take a closer look at the field $changeProof$. It uses the structure *TCPA_CHANGEAUTH_VALIDATE*. This structure is defined as follows [TCPA02]:

```
typedef struct td TCPA_CHANCEAUTH_VALIDATE {
    TCPA_SECRET newAuthSecret;
    TCPA_NONCE n1;
 } TCPA_CHANCEAUTH_VALIDATE;
```

After receiving *Message 10*, the object owner cannot validate the integrity of the message since he cannot recreate the HMAC. Additionally, the *changeProof* certificate does not include enough information to verify more than the fact that the authorisation value of an object has been changed to the value stored in *TCPA_SECRET* and that this protocol change did include the nonce $n1$ (*nonceOdd*).

To obtain a suitable *Casper* description we applied the same simplifications as in the initial AACP. See chapter 8 for more details.

11.3.2 Results

FDR provided us with various traces that violated the specifications. Rather than explaining one trace we will directly generalise the attacks found and apply it onto our DRM scenario afterwards. The interested reader can obtain one example trace from the appendix A.

Basic attack The traces found by FDR is facilitating the fact that the integrity of the second command *TPM_ChanceAuthAsymFinish* is not properly guaranteed. The object owner must rely on the good will of the platform owner because only he has the *parentKey.usageAuth* that is necessary to generate or verify the HMAC, which is responsible for the integrity of *Message 10*. Thus, the dishonest owner could intercept message 8 and inject his own authorisation secret (exchanges *encNewAuth* and *newAuthLink*).

In parallel, he could launch another AACP session with the TPM and submit the intercepted *Message 9* unchanged to the second AACP session. Therefore, the TPM would perform two asymmetric authorisation changes upon the same object — once with the faked secret of the dishonest owner and once with the secret of the object owner. The dishonest owner could intercept both replies from the TPM (*Message 10*) and forge a new *Message 10* in order to submit that message to the object owner. The faked response could contain the data object that can only be accessed via the faked authorisation secret and the *changeProof* that was generated to certify that the authorisation secret of the object has been altered to the value the object owner has provided. Since the object owner cannot verify the keyed hash sum of *Message 10* (*HMAC(parentKey.usageAuth, data)*) and the *changeProof* certificate does not include identities, he cannot discover that the certificate *changeProof* does not belong to the encrypted object in the message (*outData*).

Extended attack If we apply this attack to our scenario, we obtain the following:

1. The service provider generates a secure object on the platform of the dishonest user.

2. The service provider initiates an AACP session to change the authorisation secret to a value that is unknown to the user.

3. The dishonest user uses the AACP attack to fool the service provider into accepting the object with the faked authorisation secret. On behalf of the service provider, the trusted agent stores the key object on the trusted platform and stores the key handle inside his code base.

4. The trusted agent initialises the key object with his private key and removes that key from his code base.

5. The dishonest user uses his authorisation secret to change the PCR setting of the key object so that it can be accessed even if the system is not in a trusted state.

6. The dishonest user triggers the procedure for accessing the protected content, thereby launching the trusted agent of the service provider.

7. The platform itself is in an un-trusted system state at that stage. The trusted agent acts according to our DRM protocol and requests the relevant authorisation secret.

8. The dishonest user injects his own authorisation secret. The trusted agent relies on the fact that, if he receives the right authorisation secret and the key object is accessible, then the platform is in a desired state.

9. The trusted agent accepts the secret, loads the object and requests the TPM to access the key object.

10. The TPM verifies the object, the authorisation secret and the system state and grants access to the object.

11. The trusted agent obtains his private key and decrypts the symmetric key to access the protected data. content.

11.3.3 Discussion

If one assumes that the trusted platform is a single user system, this attack clearly does not exist. However, the specification states that the AACP was introduced for cases in which the ADCP the owner of the platform (owner of the parent object) could overhear the conversation. Therefore, he could deduce the new secret. This only makes sense if there are more users or agents involved. We are aware that if we would change the overall structure of our DRM protocol it is possible to counterbalance the AACP flaw. An easy way to prevent the attack would be to include the identity of the AACP initiators (service provider and platform owner) in the *changeProof* certificate.

We have only focused our analysis on the AACP part. The complete DRM protocol, however, is comprised of many sequentially executed protocols. This, as discussed in chapter 9 means, for a thorough analysis, we also have to verify whether or not there are feature interactions between the protocol runs that could lead to an illicit outcome. Since we made the most restrictive assumptions about all the protocol parts except the AACP, we are pretty confident that this is not the case in our little example. However, to tackle similar and more realistic examples, it is necessary to develop techniques that can guarantee or even enforce that there are no unintended interactions between the protocol runs. As already discussed in 9 this is an avenue for future research.

11.4 Conclusion

In this chapter we have briefly introduced digital rights management and some issues that have to be solved. We used TCPA technology to suggest a simple DRM protocol.

The approach used the AACP protocol from chapter 8. In the authorisation chapter we only included two participants - the owner of the platform and the platform itself. We used the information in [TCPA02] to extend the protocol so that it involved the intended three participants: the owner, the remote entity and the trusted platform. Whilst the vulnerability discovered in chapter 8 did not directly affect the functionality of the protocol, it did so in the version with three participants. We discussed how a dishonest user could circumvent the DRM protection that was based on the AACP. Finally, we discussed the seriousness of the discovered flaw.

Note this protocol did not include the general management of of the digital content's usage rights it is trying to protect. Introducing a holistic solution that solves current problems within the DRM area is left for future research.

Chapter 12

Conclusion

12.1 Summary

In this thesis we have shown that general-purpose model checkers such as FDR can be used to analyse intrusion detection systems, trusted platforms and their environment.

Current testing methods to evaluate IDSs are not sufficient for meeting the increasing needs for security. They use predefined attack scripts to verify whether the IDS is working properly. These approaches seem favourable on first sight since they are not difficult to set up and the results are easy to interpret. However most of them suffer from one or more of the following problems: they are largely unable to find more complex attacks such as emergent faults, they are usually unable to find new attacks and they are unable to verify designs. The last point is, in early stages of the development of security critical systems, of utmost importance.

Trusted platforms seek to establish chains of trust between processes. These processes do not necessarily reside on one host. Thus protocols are required that link these processes together. This situation increases the complexity and difficulty of verifying the trusted platforms.

Our goal was to find new ways to verify the detection capability of IDSs and to use CSP to analyse the trusted platform concept of the TCPA. The results can be summarised as follow:

Investigation whether CSP is suitable to model intrusion detection infrastructures and trusted platforms. In the IDS's case, we did so by first considering the reproduction of known attacks. Once we reached that goal, we used our knowledge to broaden the focus and to decrease the restrictions of our models. In section 3.3, we considered whether the Internet Protocol version 4 (IPv4) gives an attacker the opportunity to launch an undetected attack against a target. In section 3.4 we increased the complexity of our CSP model. The model from now on considers all, for packet reassembly, relevant fields. Section 3.5 describes a non-deterministic process based on RFC 815, thus enabling us to verify the in-

teractions between network nodes that use different reassembly algorithms. For the first two models we also suggested improvements that prevent the occurrence of the detected vulnerabilities.

To investigate whether or not CSP can be used to verify the TCPA's approach we analysed the built-in authorisation protocols, the boot sequence, the context management and an appropriate DRM setting. In chapter 8 we showed how one can use the approach presented in [BL02] to verify a stream authentication protocol. Moreover we discussed the seriousness of the discovered weaknesses and their relevance for future protocols. Chapter 9 discusses not only the session caching process of the TPM but also ways how one can reduce the complexity of external communication protocols. In chapter 10 we presented a CSP model of the boot sequence and its logging process. In addition, we discussed various weaknesses of the model and their implications on the commercial market. Finally in chapter 11 we extended one of the authorisation protocols in an attempt to design a DRM protocol.

Development of simplifications that reduce the unmanageable state space of our models. Intrusion detection systems have to guard events within a network, therefore these systems are usually very complex. This complexity is further increased by the degree of distribution of the network under surveillance. If the network is highly distributed many agents have to be employed to oversee the system. These agents have to communicate with other agents to share vital information. The complexity is additionally increased by more elaborate attack techniques. Hence the IDS and the environment it is embedded in can only be modelled accurately by a process that has very many states.

Unfortunately, since FDR scans through the whole state space of its models and the data complexity of the model is above a certain threshold, not necessarily infinite, it becomes unfeasible to use FDR. In order to meet this requirement we pruned away certain fields and functionality of IPv4. Further we restricted the network topology and the scope of every involved data type.

However by applying these abstractions it remained unclear whether these restrictions did not hide vulnerabilities. In section 4 we showed that detail we pruned away did not harbor additional attacks.

We changed the focus of the time-to-live model from section 3.3, to move towards a more complete analysis, now only requiring $\{A, B\}$ as potential bit sequences. Finally, we generalised the parameters of the system further. We showed, by emloying data independence techniques [Laz97], that a buffer of size 5 and the range $\{1..3\}$ for the TTL field are sufficient. Finally we discussed the generalisation of the network topology.

Investigation of time in relation to intrusion detection. We inspected not only different ways to model timeout mechanisms within the CSP calculus but also derived from our results a method to convert untimed processes that use

the timeout operator into a corresponding discrete timed process. We used two approaches, first we designed an easy-to-build CSP model, and second we introduced, a more complicated, discrete time CSP model. The first model employed a sliding choice operator to model the timeout mechanism in the IPv4 reassembly algorithm. Whilst the model was easy to develop the results obtained were ambiguous and difficult to interpret. Hence we designed a more precise system with *tock* events. This discrete timeout model was more difficult to develop however the results left no room for wrong interpretations.

This examination concluded with a discussion about the relationship between these models. Using these relations we derived a function f that transforms untimed processes into a discrete timeout process. We showed that whenever the specification is refined by an untimed model then it is also by the corresponding discrete model. This gives us the option, for untimed safety specifications, to verify the process using the simpler version of timeouts, without fearing to miss a specification violation in the more complex discrete time model.

Finding ways to reduce the complexity of external communication protocols. In chapter 9 we had to model a system that relied on external input. The focus of the investigation was not on the communication protocol but on the internal transactions that take place after receiving a certain piece of data. If one wants to verify processes within the TCPA setting we described, this scenario repeatedly occurs. In section 9.4 we discuss certain ways how one can prune away details of external communication protocols. We classified protocols according to the properties they offer and picked five classes that occur frequently:

1. authentication in combination with drop resistant stream integrity
2. authentication in combination with stream integrity
3. secrecy in combination with drop resistant stream integrity
4. secrecy in combination with stream integrity
5. authentication and secrecy in combination with drop resistant stream integrity

For every case we discussed the possible actions an intruder can perform. From them we derived a model that incorporated the most general intruder and finally, we showed that our model still satisfied the required security properties.

12.2 Related work

Vulnerability identification and analysis has been a key topic in computer science for some time now. One of the first attempts to build an automatic vulnerability

detector was COPS [FS90]. In common with other approaches (SATAN, TIGER and NESSUS [Nes]) it looks for already known attacks. It manages this by firing known attack patterns against a particular host. However, this kind of direct testing is not suitable for spotting unknown attacks.

[LFG+00a, LFG+00b, PN98] used synthetic or real world attack sets to verify whether or not IDSs are able to detect the attacks. However, since the test sets contain either known or designed, synthetic attacks, by human hands [RDS99], they are not able to detect completely novel weaknesses.

In [RA00], Ritchey and Ammann propose a high level approach to detect whether it is possible for an attacker to leverage the existing vulnerabilities in the network to get a higher degree of access. The configurations of the network nodes were abstracted to state machines and the attacks were represented as state transitions in these machines.

In contrast to that [RS02] uses a low level description of a UNIX file system to spot single configuration vulnerabilities. The focus is more on finding new attacks rather then finding a combination of attacks that enables the attacker to penetrate the system even more. The differences between ours and Sekar's approach lies in the state space: they use an infinite model and we use approximation to restrict our model to a finite one. Therefore, we can use a common model checker such as FDR. Additionally they use invariants to manifest their security policy in contrast to using a specification, as we do. The difference becomes clear during the examination of the counter examples: they have to establish an intentions model to prune away all paths that are against the defined invariant but do not violate the security policy — we have no such paths.

Another approach of using model checkers for vulnerability identification is proposed in [AB00, ADX00]. However they use the model checker combined with mutation testing techniques to generate and recognise test sets rather then testing directly.

[RB01] was using our paper [RL02] as a foundation to show that the language TLA+ can be used to achieve the same as CSP. Furthermore, they showed how to use TLA+ to introduce time into our model. In comparison to our approach (completed early 2003) they do not distinguish between different ways how one can model time.

The topic of trusted computing is a frequently discussed topic in today's media [Him02, Pfi02, Plu02, Plu02b, And03]. At the beginning the majority of published work was very negative about the TCPA's and Microsoft's approach [Him02, Pfi02, Plu02, Plu02b]. The main arguments against the architecture were: loss of privacy and loss of control of the own computer platform. A great portion of the justifications that were used were based on wrong understanding of the design descriptions. This was only partly the fault of the the media, since the TCPA's initial descriptions were very ambiguous. This has improved since the beginning of 2002. However even now we faced serious restrictions because of poor descriptions of mandatory functionality. This, as shown in 3.5, is can lead

to vulnerabilities in the system's final implementation.

Since the TCPA's vision of an extendable trusted platform system is still very young there are no formal verifications that analyse this set of features (as far as the author is aware of). Clearly, there are many papers that describe and discuss the specification [TCPA03d, TCPA02], such as [Pea02] or [Mh04] but they all fall short on formally justifying the weaknesses or advantages they highlight.

On the other hand there are many publications that relate to the approaches we used to verify the parts of the trusted platform; for instance [LR97, RSG+01, LBH01] concern with the formal verification of security protocols using model checkers. Broadfoot and Lowe in [BL02] build the foundation for our analysis of the stream authentication protocol OS-AP (and OI-AP). They analyse the time efficient loss-tolerant authentication protocol which uses key chains to bind the messages of a data stream together. [BL03] investigates the relation ship between the secure transport layer protocol (SSL) and a high level protocol that is using it. They suggest how one can abstract away a low level protocol that provides authentication, secrecy and integrity. In section 9.4 we extend their work by introducing the distinction between stream and message integrity. We discuss 5 different models that allow to prune away the transaction overhead that is needed in order to guarantee the properties of the low level protocol.

The advantages of the approach we use to verify IDSs and the TCPA architecture are:

- The approach finds all possible attacks subjected to the modelling assumptions, not just known ones, due to the working principle of our model checker FDR.
- FDR provides us with easy-to-understand counter examples.
- The traces can be converted into signatures or profiles (IDS only).
- Due to the modularity of our models, the workload to add new processes or to change the network in our intrusion detection verification is low.
- We use a finite model, which allows us to use common model checkers.
- Our testing is specification based, which keeps the effort of encoding security policies low.

12.3 Future work - IDS

As shown so far, the CSP approach is a suitable way for identifying emergent faults. We will explore some new protocols and their impact on the intrusion detection area. IPv6 [CEC02, HD98b, KA98a, KA98b, KA98c, MD99, MDM96] is the upcoming protocol, so we will model and analyse it.

Since attacks are getting more sophisticated (e.g., distributed [CIA99, Gib02]) more complex systems structures of IDSs are required to collect and interpret expressive data [JM00, BR, Hel00]. If we consider other security policy enforcing devices such as firewalls, the verification of the overall systems seem to be unmanageable. [JW01] uses formal methods to analyse the behaviour of firewall rule sets. The extension of this and our work would include the verification of the whole security perimeter. Considering the size of out current IDS models and the size of the model in [JW01] this seems to be unmanageable at first sight. However, it should be possible by employing the following three strategies: using more layers of abstraction; data-independence techniques to restrict scopes of identifiers; as well as using data independent induction [CR00, CR99] to restrict the complexity of arbitrary network topologies. The last two methods are well explored [RL03] and it should be a feasible task to adapt them accordingly. However, even combining these three tasks may not be sufficient to cover all the interactions within the systems. Thus, by inspecting only particular modules of the overall system, hidden interactions between features [RL02] can cover new vulnerabilities.

One could use the Ritchey / Ammann approach [RA00] as a high-level representation and the approach described in this thesis as a low-level foundation. First we would be able, by using FDR, to retrieve the possible weaknesses on a process level. Afterwards we could use the [RA00] approach to verify whether these vulnerabilities can be used to penetrate the system even more. In other words one can use [RA00] as a vulnerability interactions verifier. The remaining work would be to build a bridge between these two layers. To do so we could use our traces to generate an attack graph [JSW02, Sch99, HWS+01] the attack graph then could be used to generate the Ritchey / Ammann model. [RA00] uses a natural language to describe the vulnerabilities and relations between the network nodes. Using the attack graph approach would have the advantage that we would obtain an unambiguous specification of the overall system with all its attack possibilities. As an extension towards this end one could design a algorithm that uses our formal description to generate the high-level model automatically.

Another problem that has been mentioned earlier, which remains unaddressed in this thesis, is concerned with the automatic generation of signatures or profiles of IDSs. An algorithm has to be designed that uses the structure of a simulated intrusion detection environment and the resulting traces to generate universal signatures that encompass all possible variations of the trace provided. Since FDR theoretically finds not only one instantiation of a specific attack but also all variations, the algorithm has to correlate all retrieved traces to more generic super classes. With these super classes it should then be possible to derive proper signatures. Using formal methods to derive signatures or profiles was already proposed in [AD03] and [MRS01], however their approaches were by far not free from flaws and more of a theoretical nature, thus only usable in a highly theoretical environment.

In another vein, through the long term it is quite unsatisfying only to be capable of verifying IDSs that are based on signature detection. Therefore we have to solve the following problems:

- How can we classify and represent vulnerabilities and attacks? We have to find a better way than representing an attack as a bit pattern, because some IDSs are not signature based.
- How does this technique apply to industrial-scale problems? Usually model checking can only be applied to small or medium sized problems. A multi layer abstraction framework needs to be established to address large-scale problems. One possibility could be to close the gap between the Ritchey and Ammann [RA00] approach and ours. Hence it would be possible to evaluate the interactions between single modules on a low level and to convert the obtained results into a high level model to analyse the relations between the spotted vulnerabilities.
- How can we abstract a network or a network node without losing too many details? As mentioned before the problem with abstraction is that one may lose important detail. It would be of great use to design a technique that allows us to prove whether or not the lost detail was important.

12.4 Future work - TCPA

In this thesis we have shown that CSP and various abstraction techniques allow us to verify certain processes of the TCPA architecture. In the future we aim to model more functionality of the TPM. So far we have not included the operating system. Especially if we want to extend our work to cover the NGSCB (if a proper specification is available), we have to include more details about the software state of the trusted platform. Towards this end we have to develop techniques that can determine whether unintended feature interaction between certain sub modules takes place. Closely related to that, many high level protocols consist of many sequentially executed protocols. In addition these protocols usually are to large to be verified together. Hence, in order to, realistically, determine whether the overall protocol satisfies the requirement we have to establish a framework with which we can determine whether or not the individual protocol runs harbour some hidden interactions that could enable an intruder to exploit the system.

As mentioned in the DRM chapter, the protocol we suggested fell short on many points that are important for being a realistic DRM protocol. It is a goal of future research to remedy this. Therefore we have to develop an architecture that includes the virtual machine, the rights sets and the distribution strategy. Special care will be taken to find a suitable way to represent the description language of the rights sets; so that modifying the sets is an easy task.

Besides the obvious functionality our approach should provide answers to following questions:

- What happens if one wants to transfer the digital content to another machine?
- What happens if one wants to generate a backup?
- How can a user restrict the degree of control a service provider possesses over his platform?

The DRM represents only one example of an application scenario that could profit from the TCPA's trusted platform. Other scenarios such as e-voting and general e-commerce transactions are possible areas that could be included in our investigation.

Appendix A

Appendix

Example trace for attack on the session caching mechanism.

```
receive.T.T.(Msg1,Sq.<Na2,O2>,<>)
meta_locksessionschedulerTPM
meta_startstoreAuthHandleTPM.Ah1.Na2.O2.sec2
meta_startsessioncountMetaTPMSessionManager1
meta_readSessionCounterMetaSessionStorage.0
meta_finishsessioncountMetaTPMSessionManager1.0.Ah1.Ah1
meta_writeSessionParallel2MetaTPMSessionManager2.1.Ah1.Na2.O2.sec2
meta_increaseSessionCounterMetaTPMSessionManager2
meta_finishstoreAuthHandleTPM
meta_unlocksessionschedulerTPM
meta_locksessionschedulerTPM
meta_getNonceNonceManagerB.Nt1
send.T.T.(Msg2,Sq.<Ah1,Nt1>,<>)
meta_unlocksessionschedulerTPM
receive.T.T.(Msg1,Sq.<Na2,O2>,<>)
meta_locksessionschedulerTPM
meta_startstoreAuthHandleTPM.Ah2.Na2.O2.sec2
meta_startsessioncountMetaTPMSessionManager1
meta_readSessionCounterMetaSessionStorage.1
meta_readSessionSpaceMetaSessionStorage.1.Ah1
meta_finishsessioncountMetaTPMSessionManager1.1.Ah1.Ah1
meta_startTPM_SaveAuthContextMetaTPMSessionManager2.Ah1
int_startTPM_SaveAuthContextExternalSessionManager3.Ah1
meta_readSessionSpaceTPM_SaveAuthContext.1.Ah1
meta_readSessionParallelTPM_SaveAuthContext.Ah1.Na2.O2.sec2
int_ResetSessionTPM_SaveAuthContext.Ah1
meta_decreaseSessionCounterTPM_SaveAuthContext
int_finishTPM_SaveAuthContextExternalSessionManager3.Storage.
                                                (Ah1,Na2,O2,sec2)
meta_storeSessionExtSessionStorageExternalSessionManager2.N.
```

```
                                          Storage.(Ah1,Na2,O2,sec2)
meta_finishTPM_SaveAuthContextMetaTPMSessionManager2
meta_writeSessionParallel2MetaTPMSessionManager2.1.Ah2.Na2.O2.sec2
meta_increaseSessionCounterMetaTPMSessionManager2
meta_finishstoreAuthHandleTPM
meta_unlocksessionschedulerTPM
meta_locksessionschedulerTPM
meta_startrequestAuthHandleTPM.Ah1
meta_startsessioncountMetaTPMSessionManager1
meta_readSessionCounterMetaSessionStorage.1
meta_readSessionSpaceMetaSessionStorage.1.Ah2
meta_finishsessioncountMetaTPMSessionManager1.1.Ah2.Ah2
send.O.T.(Msg1,Sq.<Na2,O1>,<>)
meta_deleteSessionExtSessionStorage.1
meta_injectSession.Storage.(Ah1,Na2,O1,sec2)
meta_startTPM_LoadAuthContextMetaTPMSessionManager1.Ah1
meta_storeSessionExtSessionStorageExternalSessionManager2.N.Storage.
                                                  (Ah1,Na2,O1,sec2)
meta_retriveSessionExtSessionStorageExternalSessionManager4.Ah1.
                                          Storage.(Ah1,Na2,O1,sec2)
meta_startsessioncountExternalSessionManager4
meta_readSessionCounterMetaSessionStorage.1
meta_readSessionSpaceMetaSessionStorage.1.Ah2
meta_finishsessioncountExternalSessionManager4.1.Ah2.Ah2
meta_startTPM_SaveAuthContextExternalSessionManager4.Ah2
int_startTPM_SaveAuthContextExternalSessionManager3.Ah2
meta_readSessionSpaceTPM_SaveAuthContext.1.Ah2
meta_readSessionParallelTPM_SaveAuthContext.Ah2.Na2.O2.sec2
int_ResetSessionTPM_SaveAuthContext.Ah2
meta_decreaseSessionCounterTPM_SaveAuthContext
int_finishTPM_SaveAuthContextExternalSessionManager3.Storage.
                                                  (Ah2,Na2,O2,sec2)
meta_storeSessionExtSessionStorageExternalSessionManager2.N.
                                          Storage.(Ah2,Na2,O2,sec2)
meta_finishTPM_SaveAuthContextExternalSessionManager4
int_startTPM_LoadAuthContextExternalSessionManager4.Storage.
                                                  (Ah1,Na2,O1,sec2)
meta_readSessionCounterTPM_LoadAuthContext.0
meta_writeSessionParallelTPM_LoadAuthContext.N.Ah1.Na2.O1.sec2
meta_increaseSessionCounterTPM_LoadAuthContext
int_finishTPM_LoadAuthContextExternalSessionManager4.Ah1
meta_finishTPM_LoadAuthContextMetaTPMSessionManager1
meta_readSessionParallelMetaTPMSessionManager1.Ah1.Na2.O1.sec2
meta_finishrequestAuthHandleTPM.Ah1.Na2.O1.sec2
receive.T.T.(Msg3,Sq.<O1,Ah1>,<>)
```

```
receive.T.T.(Msg3a,Sq.<T,Na2>,<>)
receive.T.T.(Msg3b,Hash.(HMAC,<T,T,sec2,Na2,O1,Nt1,Na2>),<>)
meta_startstoreAuthHandleTPM.Ah1.Na2.O1.sec2
meta_startsessioncountMetaTPMSessionManager1
meta_readSessionCounterMetaSessionStorage.1
meta_readSessionSpaceMetaSessionStorage.1.Ah1
meta_finishsessioncountMetaTPMSessionManager1.1.Ah1.Ah1
meta_writeSessionParallel2MetaTPMSessionManager2.1.Ah1.Na2.O1.sec2
meta_finishstoreAuthHandleTPM
meta_TPMGrantAccessto.O1.T
```

Bibliography

[AB00] P. Ammann and P. Black. Test Generation and Recognition with Formal Methods. In First International Workshop on Automated Program Analysis, Testing, and Verication (WAPATV'00), pages 64–67, 2000.

[AD03] Luigi V. Mancini Antonio Durante, Roberto Di Pietro. Formal Specification for Fast Automatic IDS Training. In Formal Aspects of Security Volume 2629 / 2003, pages 191–204, 2003.

[ADX00] Paul Ammann, Wei Ding, and Daling Xu. Using a Model Checker to Test Safety Properties. 2000. ISE Department, MS 4A4, Georg Mason University, 4400 University Drive Fairfax, VA 22030 USA.

[AH04] Alapan Arnab and Andrew Hutchison. Digital Rights Management - A current review. In Technical Report CS04-04-00, Department of Computer Science, University of Cape Town, 2004. `http://pubs.cs.uct.ac.za/archive/00000114/`

[AK96] Ross Anderson and Markus Kuhn. Tamper Resistance - a Cautionary Note. The Second USENIX Workshop on Electronic Commerce Proceedings, Oakland, California, November 18-21, 1996, pp 1-11, ISBN 1-880446-83-9. `http://www.cl.cam.ac.uk/~mgk25/tamper.html`

[AKS96] T. Aslam, I. Krsul, and E. H. Spafford. Use of a Taxonomy of Security Faults. In Proc. 19th NIST-NCSC National Information Systems Security Conference, pages 551–560, 1996.

[And80] J. P. Anderson. Computer Security Threat Monitoring and Surveillance. In Tech. Rep., James P Anderson Co., Fort Washington, PA, 1980.

[And03] Ross Anderson. TCPA/Palladium-FAQ. Aug 2003. `http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html`

[Arc02] Myla Archer. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In Workshop on Issues in the Theory of Security, 2002.

[ATT99] AT & T Research. Beyont Concern: Understanding Net Users' Attitudes About Online Privacy. 1999. `http://www.research.att.com/resources/trs/TRs/99/99.4.3/report.htm`

[AXE98a] AXENT. Intrusion Detection Methodologies. Feb 11 1998.

[AXE98b] AXENT. Security Assessment Methodologies. Mar 9 1998.

[Axe00] Stefan Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Technical Report 99-15, Chalmers University, Mar 2000.

[BCP+99] Franco Bartolini, Vito Cappellini, Alessandro Piva, A. Fringuelli and Mauro Barni. Electronic Copyright Management Systems: Requirements, Players and Technologies. DEXA Workshop 1999: 896-898. `http://csdl.computer.org/comp/proceedings/dexa/1999/0281/00/02810896abs.htm`

[Bel93] Steven M. Bellovin. Packets Found on an Internet. 1993.

[Bid04] Peter N. Biddle. Next-Generation Secure Computing Base. WinHEC2004, Nov 2004.

[BL02] P.J. Broadfoot and Gavin Lowe. Analysing a Stream Authentication Protocol using Model Checking. In Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS). May 2002.

[BL03] Philippa Broadfoot and Gavin Lowe. On Distributed Security Transactions that use Secure Transport Protocols. In Proceedings of the 16th IEEE Computer Security Foundations Workshop, 2003.

[Bla98] Uyless Black. TCP/IP and Related Protocols. Computer Communications. McGraw-Hill, 1998.

[BLR00] P.J. Broadfoot, Gavin Lowe, and A.W. Roscoe. Automating Data Independence. In Proceedings of ESORICS, pages 175 – 190, 2000.

[BR] Joseph Barrus and Neil C. Rowe. A Distributed Autonomous-Agent Network-Intrusion Detection and Response System. Code cs/rp, Naval Postgraduate School, Monterey, CA 93943.

[BaRa] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. Microsoft Reasearch.

[Bro01] P. J.Broadfoot. Data Independence in the Model Checking of Security Protocols. PhD thesis, University of Oxford, University College, Trinity Term 2001.

[BW97] Andreas Bonnard and Christian Wolff. Gesicherte Verbindung von Computernetzen mit Hilfe einer Firewall. `http://www.bsi.de/literat/studien/firewall/fwstud97/fw-stud.pdf`

[CC02] Lus Caires and Luca Cardelli. A Spatial Logic for Concurrency (Part I). IC special issue on TACS'01, 2002.

[CD98] A. Conta and S. Deering. RFC 2463, Internet Control Message Protocol (ICMPv6) for the Internet Protocol version 6 (IPv6) Specification. Dec 1998.

[CEC02] Commission of the European Communities. New Generation Internet - Priorities for Action in Migrating to the new Internet Protocol IPv6. COM(2002) 96 final:15, Feb 2002.

[CER02] CERT/CC. Code Red Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. In CERT Advisory CA-2001-19, 2002.

[CER03] Coordination Center CERT. Analysis Console for Intrusion Databases. 2003. `http://www.cert.org/kb/acid/`

[Che99] Cheskin. Research and Studio Archetype. In eCommerce Trust Study. `http://www.sapient.com/cheskin/`

[CIA99] CIAC. Distributed System Intruder Tools Trinoo and TFN. CIAC 00.040, Dec 21 1999.

[CIDF] Common Intrusion Detection Framework (CIDF). `http://www.isi.edu/gost/cidf/`

[Cis04] Cisco IDS. 2004. `http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/index.shtml`

[CJPL02] Amy Carroll, Mario Juarez, Julia Polk, and Tony Leininger. Microsoft "Palladium": A Business Overview, Combining Microsoft Windows Features, Personal Computing Hardware, and Software Applications for Greater Security, Personal Privacy and System Integrity. Microsoft Windows Trusted Platform Technologies (formerly Content Security Business Unit),August 2002.

[Cla82] David D. Clark. RFC 815 IP Datagram Reassembly Algorithms. MIT Laboratory for Computer Science Computer Systems and Communications Group, Jul 1982.

[Cli04] Cliff. Linux and DRM?. Feb 2004. `http://slashdot.org/article.pl?sid=04/02/10/2329229&mode=thread`

[CW96] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, 28 (4), Dec 1996.

[CR99] S. Creese and A. Roscoe. Formal Verification of Arbitrary Network Topologies. In Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99). CSREA Press, 1999.

[CR00] S. Creese and A. Roscoe. Data Independent Induction over Structured Networks. In International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '00), Las Vegas, USA, June 2000.

[CS02] Sadie Creese and William Simmonds. Specification and Verification of Selected Intrusion Tolerance Properties using CSP and FDR. MAFTIA deliverable D7 IST-1999-11583, QuinetiQ (UK), Feb 2002.

[CZC00] D. Brent Chapman, Elizabeth D. Zwicky, and Simon Cooper. Building Internet Firewalls. O'Reilly, Jun 2000. ISBN: 1-56592-871-7.

[DA99] T. Dierks and C. Allen. RFC 2246 - The TLS Protocol Version 1.0. Jan 1999. `http://www.faqs.org/rfcs/rfc2246.html`

[Dav93] J. W. M. Davies. Specification and Proof in Real-Time. Cambridge University Press, 1993.

[Den87] D. E. Denning. An Intrusion-Detection Model. In IEEE Transactions on Software Engineering, SE-13, pages 222–232, 1987.

[Dit05] Dave Dittrich. Distributed Denial of Service (DDoS) Attacks/tools. Jan 2005 . `http://staff.washington.edu/dittrich/misc/ddos/`

[dR81] Marina del Rey. RFC 791 Internet Protocol: DARPA Internet Program Protocol specification, 1981.

[dR81b] Marina del Rey. RFC 793 Transmission Control Protocol DARPA Internet Program Protocol Specification. Sep 1981.

[DY83] D. Dolev and A. C. Yao. On the security of public-key protocols. In Communications of the ACM, 29(8):198-208, Aug 1983.

[Ega75] J. Egan. Signal Detection Theory and ROC Analysis. In New York: Academic, 1975.

[Ein01] Nathan Einwechter. An Introduction to Distributed Intrusion Detection Systems. `http://www.securityfocus.com`, Jan 8 2001.

[Els00] David Elson. Intrusion Detection, Theory and Practice. `http://www.securityfocus.com`, Mar 27 2000.

[FS90] Daniel Farmer and Eugene H. Spafford. The COPS Security Checker System. In USENIX Summer, pages 165–170, 1990.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate Abstraction for Software Verification. Compaq Systems Research Center, 103 Lytton Ave, Palo Alto, CA 94301; 2002.

[GGH+00] Paul Gardiner, Michael Goldsmith, Jason Hulance, David Jackson, A.W. Roscoe, and Bryan Scattergood. FDR2 User Manual. Formal Systems (Europe) Ltd., 2000.

[Gib02] Steve Gibson. Distributed Reflection Denial of Service ,Description and Analysis of a Potent, Increasingly Prevalent, and Worrisome Internet Attack. Feb 2002. `http://www.grc.com/dos/drdos.htm`

[GJ88] C. A. Gunter and A. Jung. Coherence and Consistency in Domains. Third Annual Symposium on Logic in Computer Science, Computer Society Press of the IEEE, 1988. `citeseer.ist.psu.edu/gunter90coherence.html`.

[Gro03] Jeff Grove. Legal and technological efforts to lock up content threaten innovation. In Communications of the ACM. `http://doi.acm.org/10.1145/641205.641222`

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. 9th International Conference on Computer Aided Verification (CAV'97), 1997. `citeseer.ist.psu.edu/graf97construction.html`.

[GU00] Patricia Gilfeather and Todd Underwood. Fragmentation made friendly. `http://www.cs.unm.edu/~maccabe/SSL/frag/FragPaper1/Fragmentation.html`, Jan 31 2000.

[Hac00] Eric Hacker. Re-Synchronizing a NIDS. `http://www.securityfocus.com`, Sep 22 2000.

[Hac01] Eric Hacker. IDS Evasion with Unicode. `http://www.securityfocus.com`, Jan 03 2001.

[HD98a] R. Hinden and S. Deering. RFC 2373, IP Version 6 Addressing Architecture. Jul 1998.

[HD98b] R. Hinden and S. Deering. RFC 2460 Internet Protocol, Version 6 (IPv6) Specification. Dec 1998.

[Hei03] Stephen Heil. Windows Trusted Platform and Infrastructure Information Newsletter. Windows Trusted Platform and Infrastructure Team, Apr 2003.

[Hel00] G. Helmer. Intelligent Multi-Agent System for Intrusion Detection and Countermeasures. Phd thesis, Iowa State University, Ames, IA, USA, Dec 2000.

[HF00] Steven A. Hofmeyr and Stephanie Forrest. Architecture for an Artificial Immune System. Evolutionary Computation, 8(4):443–473, 2000.

[HG00] Greg Hoglund and Jon Gray. Multiple Levels of De-Synchronization and other Concerns with Testing an IDS. `http://www.securityfocus.com`, Aug 2000.

[Him02] Gerald Himmelein. Der digitale Knebel, Intel und Microsoft wollen Daten vor dem Anwender schtzen. In C't, 2002.

[HK98] Josef van Helden and Stefan Karsch. Grundlagen, Forderungen und Marktbersicht fr Intrusion Detection Systeme (IDS) und Intrusion Response Systeme (IRS). Oct 1998. `http://www.bsi.de/literat/studien/ids/ids-stud.pdf`

[HL01] Mei Lin Hui and Gavin Lowe. Fault-preserving simplifying transformations for security protocols. In Journal of Computer Science, 9(1, 2):3-46, 2001.

[Hoa78] C. A. R. Hoare. A Model for Communicating Sequential Processes. Communications of the ACM, 21:666–677, 1978.

[Hoa85] C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985. ISBN: 0-13-153271-5.

[How97] John D. Howard. An Analysis of Security Incidents on the Internet 1989–1995. In PhD Thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1997.

[HSTL90] K. Chen Henry S Teng and S. C. Lu. Security Audit Trail Analysis using Inductively Generated Predictive rules. In Proceedings of the 11th National Conference on Artificial Intelligence Applications, pages 24–29, 1990.

[HWS+01] Guy Helmer, Johnny Wong, Mark Slagell, Vasant Honavar, Les Miller, and Robin Lutz. A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System. 1st Symposium on Requirements Engineering for Information Security Indianapolis, 2001.

Department of Computer Science, 226 Atanasoff Hall, Iowa State University.

[ID93] C.N. Ip and D.L. Dill. Better Verification Through Symmetry. Computer Hardware Description Languages and their Applications, Elsevier Science Publishers B.V., Amsterdam, Netherland, 1993. `citeseer.ist.psu.edu/article/ip93better.html`.

[Ilg93] Koral Ilgun. USTAT: A Real-Time Intrusion Detection System for UNIX. In Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy, pages 16–28, Oakland, CA, 1993.

[Inn01] Paul Innella. The Evolution of Intrusion Detection Systems. `http://www.securityfocus.com`, Nov 2001.

[Ip96] Chung-Wah Norris Ip. State Reduction Methods For Automatic Formal Verification. Ph.D. Thesis Stanford, Dec 1996.

[ISS05] Internet Security Systems. Proventia Intrusion Detection. `http://www.iss.net/products_services/enterprise_protection/proventia/a_series.php`

[JH03] Joe McAlerney James Hoagland, Stuart Staniford. SnortSnarf. In SnortSnarf Handbook, 2003. `http://www.snort.org/dl/contrib/data_analysis/snortsnarf/`

[JM00] J. Allen J. McHugh, A. Christie. Defending Yourself: The Role of Intrusion Detection Systems. In IEEE Software 17, pages 42–51, 2000.

[JW01] Jan Jrgens and Guido Wimmel. Specification-Based Testing of Firewalls. 2001. `http://www4.in.tum.de/~juerjens/research.html`.

[JSW02] S. Jha, O. Sheyner, and J. Wing. Two Formal Analyses of Attack Graphs. Journal of Computer Security, 10:49 – 63, 2002.

[KA98a] S. Kent and R. Atkinson. RFC 2401, Security Architecture for the Internet Protocol. Nov 1998.

[KA98b] S. Kent and R. Atkinson. RFC 2402, IP Authentication Header. Nov 1998.

[KA98c] S. Kent and R. Atkinson. RFC 2406, IP Encapsulating Security Payload (ESP). Nov 1998.

[KB] J. Kim and P. Bentley. The Artificial Immune Model for Network Intrusion Detection. `http://www.dcs.kcl.ac.uk/staff/jungwon/publication.html`

[KB99] J. Kim and P. Bentley. The Human Immune System and Network Intrusion Detection. In 7th European Congress on Intelligent Techniques and Soft Computing (EUFIT '99), Aachen, Germany, September 13-19, 1999.

[KB01] Jungwon Kim and Peter J. Bentley. An Evaluation of Negative Selection in an Artificial Immune System for Network Intrusion Detection. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), pages 1330–1337, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann.

[KCB97] H. Krawczyk, R. Canetti and M. Bellare. HMAC: Keyed-Hashing for Message Authentication. Feb 1997. `http://www.faqs.org/rfcs/rfc2104.html`

[Ken01] Frederick Karen Kent. Network Intrusion Detection Signatures — Part 1. `http://www.securityfocus.com`, 2001.

[Kes00] Gregory Kesden. Lecture 33 (Wednesday, December 6, 2000). In Course: 15-412 Operating Systems: Design and Implementation. `http://www-2.cs.cmu.edu/~dst/DeCSS/Kesden/`

[KFL94] Calvin Ko, George Fink, and Karl Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. Technical report, Department of Computer Science, University of California at Davis, 1994.

[Ko96] Calvin Ko. Execution Monitoring of Security Critical Programs in a Distributed System: A Specification-Based Approach. PhD thesis, Department of Computer Science, University of California at Davis, 1996.

[Krs98] I. Krsul. Software Vulnerability Analysis. In PhD thesis, Purdue University, West Lafayette, Indiana, 1998.

[KS94] S. Kumar and E. H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. Proceeding of the 17th National Computer Security Conference, pages pp. 11–21, Oct 1994. Baltimore, MD, USA.

[Kum95] S. Kumar. Classification and Detection of Computer Intrusions. Phd thesis, Purdue University, West Lafayette, IN, USA, Aug 1995.

[LA00] Brian Laing and Jimmy Alderson. How To Guide-Implementing a Network Based Intrusion Detection System. Internet Security Systems, 2000.

[LABW92] Lampson, Abadi, Burrows and Wobber. Authentication in Distributed Systems: Theory and Practice. In ACM0734-2071/92/1100-0000, 1992.

[Laz97] Ranko Lazić. A Semantic Study of Data-Independence with Applications to the Mechanical Verification of Concurrent Systems. D.Phil., Oxford University, 1997.

[LBH01] Gavin Lowe, Philippa Broadfoot and Mei Lin Hui. Casper - A Compiler for the Analysis of Security. Dec 2001.

[Lev95] N. G. Leveson. Safeware: System Safety and Computers. Addision-Wesley, Reading, MA, USA, 1995.

[LFG+00a] Richard Lippmann, David Fried, Isaac Graf, Joshua Haines, Kristopher Kendall, David McClung, Dan Weber, Seth Webster, Dan Wyschogrod, Robert Cunningham, and Marc Zissman. Evaluating Intrusion Detection Systems: The 1998 DARPA off-line intrusion detection evaluation. In Proceedings of the DARPA Information Survivability Conference and Exposition, Los Alamitos, CA, 2000. IEEE Computer Society Press.

[LFG+00b] Richard Lippmann, David Fried, Isaac Graf, Joshua Haines, Kristopher Kendall, David McClung, Dan Weber, Seth Webster, Dan Wyschogrod, Robert Cunningham, and Marc Zissman. The 1999 DARPA off-line Intrusion Detection Evaluation. In RAID 2000, LNCS No. 1907, Springer Verlag, New York, 2000.

[Lin04] Pete Lindstrom. Truth and Fiction about Microsoft's 'Palladium'. Mar 2004. `http://www.csoonline.com/analyst/report2317.html`

[LNY+00] Wenke Lee, Rahul A. Nimbalkar, Kam K. Yee, Sunil B. Patil, Pragneshkumar H. Desai, Thuan T. Tran, and Salvatore J. Stolfo. A Data Mining and CIDF based Approach for Detecting Novel and Distributed Intrusions. Lecture Notes in Computer Science, 2000.

[LR97] Gavin Lowe and A.W. Roscoe. Using CSP to Detect Errors in the TMN Protocol. Aug 6 1997.

[LS98] Wenke Lee and Salvatore Stolfo. Data Mining Approaches for Intrusion Detection. In Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, 1998.

[Lun90] T. F. Lunt. IDES: An Intelligent System for Detecting Intruders. In Proceedings of the Symposium: Computer Security, Threat and Countermeasures, 1990.

[Lun93] Teresa F Lunt. A survey of Intrusion Detection Techniques. In In Computer Security, pages 405–418, 1993.

[Lyo01] Gordon Lyon. The Internet Marketplace and Digital Rights Management. National Institute for Standards and Technology, Jun 2001. http://www.itl.nist.gov/div895/docs/GLyonDRMWhitepaper.pdf

[MD99] P. Marques and F. Dupont. RFC 2545, use of BGP-4 Multiprotocol Extensions for IPv6 Inter-Domain routing. Mar 1999.

[MDM96] J. McCann, S. Deering, and J. Mogul. RFC 1981, path MTU discovery for IP version 6. Aug 1996.

[MHB03] Deirdre K. Mulligan, John Han and Aaron J. Burstein. How DRM-based content delivery systems disrupt expectations of "personal use". In DRM '03: Proceedings of the 2003 ACM workshop on Digital rights management. http://doi.acm.org/10.1145/947380.947391

[mic03] michael. DRM From the Viewpoint of the Electronic Industry. Slashdot. http://slashdot.org/article.pl?sid=03/11/25/1821218&mode=thread&tid=126&tid=141&tid=188

[Mic02] Microsoft. MS Palladium Initiative - Technical FAQ. August 2002. http://www.microsoft.com.

[Mic03] Microsoft. NGSCB: Trusted Computing Base and Software Authentication. 2003. http://www.microsoft.com/resources/ngscb/documents/ngscb_tcb.doc

[Mic03a] Microsoft. Hardware Platform for the Next-Generation Secure Computing Base. 2003. http://www.microsoft.com/resources/ngscb/documents/NGSCBhardware.doc

[Mic03b] Microsoft. Security Model for the Next-Generation Secure Computing Base. 2000. http://www.microsoft.com/resources/ngscb/documents/NGSCB_Security_Model.doc

[Mic04] Microsoft. Rights Management Services in Windows. Aug 2004. http://www.microsoft.com/technet/prodtechnol/windowsserver2003/technologies/rightsmgmt/default.mspx

[Mic05] Microsoft. Next-Generation Secure Computing Base. 2005. `http://www.microsoft.com/resources/ngscb/archive.mspx`

[MRS01] John A. Marin, Daniel Ragsdale, and John Surdu. A Hybrid Approach to Profile Creation and Intrusion Detection. In Proc. of DARPA Information Survivability Conference and Exposition, June 12–14, 2001.

[Mh04] Michael Mhle. Analyse von TCPA. Groer Beleg, Technische Universitt Dresden, Feb 2004.

[Nes] Nessus a remote Security Scanner. `http://www.nessus.org/`.

[NIST95] National Institute of Standards and Technology. Secure Hash Standard. In FIPS PUB 180-1, April 1995. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`

[NIST01] National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). In FIPS 197, November 2001. `http://csrc.nist.gov/CryptoToolkit/aes/`

[Nor99] Stephen Northcutt. Network Intrusion Detection - An Analyst's Handbook. New Raiders, June 1999.

[Oua01] Joel Ouaknine. Discrete Analysis of Continuous Behaviour in Real-Time Concurrent Systems. Ph.D. Thesis, Oxford University, 2001.

[Par00] Jaehong Park Security Architectures for Controlled Digital Information Dissemination. In Proceedings of 16th Annual Computer Security Applications Conference (ACSAC'00), New Orleans, 11-15 Dec 2000. `http://computer.org/Proceedings/acsac/0859/08590224abs.htm`

[Pax99] Vern Paxton. BRO: A System for Detecting Network Intruders in Real-Time. Computer Networks, 31:2435–2463, 1999.

[Pea02] Siani Pearson Trusted Computing Platforms: Tcpa Technology in Context Prentice Hall, Aug 2002. ISBN: 0130092207.

[Pfi02] Andreas Pfitzmann. Unheilvolle Allianz: Hollywood und Microsoft, Interview mit dem Dresdener Informatikprofessor und Kryptoexperten Andreas Pfitzmann. In C't, 2002.

[Plu02] Michael Plura. Der versiegelte PC, Was steckt hinter TCPA und Palladium?. In C't, 2002.

[Plu02b] Michael Plura. Der PC mit den zwei Gesichtern, TCPA und Palladium - Schreckgespenster oder Papiertiger?. In C't, 2002.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Secure Networks, 1998.

[PR83] J. Postel and J. Reynolds. RFC 854 - Telnet Protocol Specification. May 1983. `http://www.faqs.org/rfcs/rfc854.html`

[Pri03] Anne Price. New Trusted Computing Group Formed to Advance the Adoption of open Standards for Trusted Computing Technologies. Portland Oregon, April 2003.

[RA00] R. Ritchey and P. Ammann. Using Model Checking to Analyze Network Vulnerabilities. IEEE Oakland Symposium on Security and Privacy, pages 156 – 165, May 2000.

[Rah91] D. G. Rahejy. Assurance Technologies: Principles and Practices. Engineering and Technology Management Series. McGraw-Hill, McGraw-Hill, New York, 1991.

[Ran01] Marcus J. Ranaum. Coverage in Intrusion Detection Systems. 2001. `http://www.itsecurity.com/papers/nfr1.htm`

[RB99] A.W. Roscoe and P.J. Broadfoot. Proving Security Protocols with Model Checkers by Data Independence Techniques. Journal of Computer Security: Special Issue CSFW12, 7 (2,3):147–190, Jul 1999.

[MB01] Peter Mell and Rebecca Bace. Intrusion Detection Systems. In NIST Special Publication on Intrusion Detection Systems, 2001.

[RB01] Slim Rekhis and Noureddine Boudriga. Formal Verification of Intrusion Detection Systems Using TLA+. In Proceedings of WITS, 2004.

[RSBC09] Denise M. Rousseau, S. Sitkin, R.S. Burt and C. Cammerer. Not So Different After All: A Cross Disciplinary View of Trust. In Academy of Management Review, Volume 23, Pages 1-12, 1998.

[RD03] Bill Rosenblatt and Gail Dykstra. Integrating Content Management with DRM: Imperatives and Opportunities for Digital Content Lifecycles. November 14, 2003. `http://www.drmwatch.com/resources/whitepapers/article.php/11655_3112011_1`

[RDS99] Brian Witten Eric Miller Robert Durst, Terrence Champion and Luigi Spagnuolo. Testing and Evaluating Computer Intrusion Detection Systems. In Communications of the ACM Vol. 42 no. 7, pages 53–61, 1999.

[RFP02] Rain Forrest Puppy. Whisker. 2002. `http://www.wiretrip.net/rfp/bins/whisker/v2.0/whisker-pr2.0.tar.gz`

[RL02] Gordon Rohrmair and Gavin Lowe. Using CSP to Detect Insertion and Evasion Possibilities within the Intrusion Detection Area. In Proceedings of BCS Workshop on Formal Aspects of Security, 2002.

[RL03] Gordon Rohrmair and Gavin Lowe. Using Data-Independence in the Analysis of Intrusion Detection Systems. In Proceedings of the Workshop on Issues in the Theory of Security (WITS '03), 2003.

[RL04] Gordon Thomas Rohrmair and Gavin Lowe. Using Data-Independence in the Analysis of Intrusion Detection Systems. In Theoretical Computer Science, 2004.

[Rob] Nina Amla Robert. Autoabs: Syntax-Directed Program Abstraction. `citeseer.ist.psu.edu/608703.html`.

[Ros98] A.W. Roscoe. Proving Security Protocols with Model Checkers by Data Independence Techniques. Proceedings, 1998 IEEE Computer Security Foundations Workshop, 1998.

[Ros98b] A. W. Roscoe. The Theory and Practice of Concurrency. Prentice Hall, 1998. ISBN: 0-13-674409-5.

[RP94] J. Reynolds and J. Postel. RFC 1700, Assigned Numbers. Oct 1994.

[RS02] C.R. Ramakrishan and R. Sekar. Model-based Analysis of Configuration Vulnerabilities. Journal of Computer Security, 10(1, 2), 2002.

[RSG+01] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and A.W. Roscoe. Modelling and Analysis of Security Protocols. Addison-Wesley, 2001. ISBN: 0 201 67471 8.

[RZT95] D. Reed, G. Ziemba, and P. Traina. RFC 1858: Security considerations for IP Fragment Filtering, 1995.

[SBS99] R. Sekar, T. Bowen, and M. Segal. On preventing Intrusions by Process Behavior Monitoring. In USENIX Intrusion Detection Workshop, 1999.

[SCH95] S. Staniford-Chen and L. T. Heberlein. Holding Intruders Accountable on the Internet. Proc. IEEE Symposium on Security and Privacy, Oakland, CA, pages pp. 39–49, May 1995.

[Sch97] Steve Schneider. Timewise Refinement for Communicating Processes. Science of Computer Programming, 1997. `citeseer.ist.psu.edu/schneider97timewise.html`.

[Sch99] Bruce Schneier. Modeling Security Threats. In Dr. Dobb's Journal, 1999.

[Sch00a] S. A. Schneider. Concurrent and Real-Time Systems: The CSP Approach. John Wiley, New York, 2000.

[Sec] Network Flight Recorder Security. 5 Choke Cherry Road Suite 200 Rockville MD 20850-4004. `http://www.nfr.com/`.

[SN98] Secure Networks Inc.. Ballista. `http://www.infoworld.com/cgi-bin/displayTC.pl?/reviews/980413ballista.htm`

[SNO] SNORT. `http://www.snort.org/`.

[SRI01] SRI. Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD). Oct 2001. `http://www.sdl.sri.com/projects/nides/`

[SRI02] SRI. Next-Generation Intrusion Detection Expert System. 2002. `http://www.sdl.sri.com/projects/nides/`

[SU] R. Sekar and P. Uppuluri. Synthesizing fast Intrusion Prevention/Detection Systems from High-Level Specifications. Master's thesis, State University of New York at Stony Brook, NY 11794.

[SU01] R. Sekar and P. Uppuluri. State, Experience with Specification-Based Intrusion Detection. Raid conference 2001, 2001. NY 11794.

[Sun96] Aurobindo Sundaram. An Introduction to Intrusion Detection. 1996. `http://www.acm.org/crossroads/xrds2-4/intrus.html`

[Sym01] Symantec. Hacktivism: Activism Gone High Tech. ARTICLE ID: 711, 2001. `http://enterprisesecurity.symantec.com/article.cfm?articleid=711\&EID=0`.

[Sys98] Internet Security Systems. Network- vs. Host-based intrusion detection. Technical report. `http://www.iss.net/support/documentation/whitepapers/index.php`. Oct 02 1998.

[Sys99] Internet Security Systems. Intrusion Detection Systems - Whitepaper. `http://www.iss.net/support/documentation/whitepapers/index.php`. 1999.

[Sys00] Internet Security Systems. Evaluating an Intrusion Detection Solution - A Strategy for a successful IDS Evaluation. `http://www.iss.net/support/documentation/whitepapers/index.php`. Aug 22 2000.

[Sys00b] Internet Security Systems. Intrusion Detection for the Millennium. In Technology Brief, 2000.

[Tan01] Matthew Tanase. The Future of IDS. `http://www.securityfocus.com`. Dec 4 2001.

[TCG05] Trusted Computing Group. Trusted Computing Group Homepage. `https://www.trustedcomputinggroup.org/home`

[TCPA00] Trusted Computing Platform Association. Building A Foundation of Trust in the PC. TCPA Whitepaper, Jan 2000. `https://www.trustedcomputinggroup.org/home`

[TCPA00b] Trusted Computing Platform Association. TCPA Security and Internet Business: Vital Issues for IT. TCPA Whitepaper, Aug 2000. `https://www.trustedcomputinggroup.org/home`

[TCPA02] Trusted Computing Group. TCG Main Specification Version 1.1b. TCPA Specification 1.1, Feb 2002. `https://www.trustedcomputinggroup.org/home`

[TCPA03] Trusted Computing Group. Design Principles. TCG TPM Specification Version 1.2, Oct 2003. `https://www.trustedcomputinggroup.org/home`

[TCPA03b] Trusted Computing Group. Trusted Platform Module Protection Profile. TCG TPM Specification Version 1.1, Jul 2002. `https://www.trustedcomputinggroup.org/home`

[TCPA03c] Trusted Computing Group. Structures of the TPM. TCG TPM Specification Version 1.2, Oct 2003. `https://www.trustedcomputinggroup.org/home`

[TCPA03d] Trusted Computing Group. TPM Commands. TCG TPM Specification Version 1.2, Oct 2003. `https://www.trustedcomputinggroup.org/home`

[TCPA03e] Trusted Computing Platform Association. TCG Software Stack Specification Version 1.1. TCPA Specification 1.1, Aug 2003. `https://www.trustedcomputinggroup.org/home`

[TCPA03f] Trusted Computing Platform Association. TCG Software Stack Specification Header File. TCPA Specification 1.1, Sep 2003. `https://www.trustedcomputinggroup.org/home`

[TCPA03g] Trusted Computing Platform Association. TCG PC Specific Implementation Specification Version 1.1. TCPA Specification 1.1 RC3, Aug 2003. `https://www.trustedcomputinggroup.org/home`

[TCPA04] Trusted Computing Group. TCG Specification Architecture Overview. TCG TPM Specification Version 1.2, Apr 2004. `https://www.trustedcomputinggroup.org/home`

[TCPA04b] Trusted Computing Group. TCG Glossary. TCG TPM Specification Version 1.2, Jul 2004. `https://www.trustedcomputinggroup.org/home`

[TZ04] TrustZone. TrustZone Technology Overview. Jul 2004. `http://www.arm.com/products/CPUs/arch-trustzone.html`

[Utt96] Bill Uttenweiler. Hackers Attack USAF Computers. July 1996. Aerospace Corporation Vandenberg AFB, CA.

[Ver01] Veridisc. Fairplay White-Paper. 2001. `http://64.244.235.240/news/whitepaper/docs/veridisc_white_paper.pdf`

[WIPO04] WIPO. MEDIUM-TERM PLAN FOR WIPO PROGRAM ACTIVITIES - VISION AND STRATEGIC DIRECTION OF WIPO. 2004. `http://www.wipo.int/about-wipo/en/dgo/pub487.htm`

[Yer96] F. Yergeau. RFC 2044 UTF-8, a Transformation Format of Unicode and ISO 10646. Oct 1996.

[ZP] Yin Zhang and Vern Paxson. Detecting Stepping Stones. Master's thesis, Department of Computer Science Cornell University Ithaca, NY 14853. `http://www.icir.org/vern/papers/stepping/index.html`.