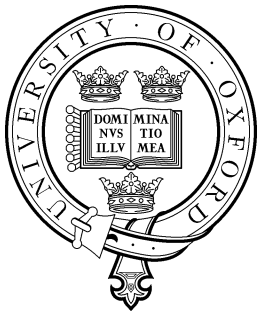
An Architectural Approach for Reasoning about Trust Properties



Cornelius Namiluko

Wolfson College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy
Computer Science

Trinity 2012

Abstract

The need for trustworthy system operation has been acknowledged in many circles. However, establishing that a system is trustworthy is a significant challenge. While trusted computing proposes technical mechanisms towards this end, less attention is directed towards providing a *basis* for trusting such systems. Consequently, it is not clear: (i) how such mechanisms influence the overall trust in a system; (ii) the properties and assumptions upon which trust is based; and (iii) the evidence necessary to reason about these properties. This can be attributed to a number of factors including: (i) the complexity of modern systems; (ii) a lack of consensus on a definition of trust; and (iii) a lack of a systematic approach for identifying and using evidence to reason about trust-related properties.

This dissertation presents research towards addressing these challenges. We argue that an architectural approach provides effective abstractions for making trust properties and assumptions explicit and reasoning about a system's ability to satisfy these properties. We propose a framework for identifying, categorising and mapping trust-properties to aspects of a system that could be used to reason about these properties. Guided by this framework, we propose and develop models for representing knowledge about a particular aspect and using it to reason about trust-properties.

A *semantic model*, based on the semantics of Z, is developed to characterise building blocks of trustworthy systems and to demonstrate how the system's constituents determine its trustworthiness. An *abstraction model* based on formal verification is developed to reason about the impact of the system's construction and configuration on its trustworthiness. Finally, two complementary models for capturing the runtime aspects of the system are developed. A *trace-based model* enables analysis of runtime evidence in the form of event logs and a *provenance-based model* captures operations on the system as a provenance graph. The models are validated on a trusted grid architecture, a password manager and a trustworthy collaborative system.

Acknowledgements

This doctoral research was made possible by funding from the Rhodes Scholarship with support from the Tony Hoare Departmental fund. I am very grateful to the Rhodes Trust and members of staff at the Rhodes House for the sponsorship and for the assistance rendered to me during my stay in Oxford. I would like to thank Sir Tony Hoare for making the departmental fund available. I am happy to report that your assistance has been well invested.

This work would not have been possible without the support of my supervisor, Dr. Andrew Martin. I would, therefore, like to give special thanks for his guidance, advice and encouragement. I am also grateful to Dr. David Wallom and members of the OeRC for sparing their time to discuss some of my work.

To my wife, Tsilazazi, and my great kids (Michelle Limpo, Jordan Tapiwa and Nick Itwi), thank you so much for the love you continue giving me and for being patient. Your patience and encouragement has surely paid off!

Thanks to my parents for the support and understanding and to all my friends (especially Lawrence Chikonde and Dr. Brian Banda) for the encouragement. You were right when you said "go on, you can do this"!

Lastly, but certainly not the least, thanks to all my colleagues in the software engineering security group, to my office mates (Dr. J. Lyle, Dr. J. Ho Huh and Dr. S. Faily), and to Dr. D. Power for comments and feedback on my work.

Contents

1 Introduction **1**

1.1 Defining Trustworthy Systems 3

1.2 The Need to Reason about Trust Properties 4

1.2.1 The Inexplicable Basis of Trust 5

1.2.2 Increased Complexity 5

1.2.3 System Dynamics 6

1.2.4 Attackers are Opportunistic 6

1.2.5 Multiple Entities Rely on the System 6

1.3 Thesis Statement 7

1.4 Overview of the Approach 7

1.4.1 Architectural Approach 7

1.4.2 Building Block Approach 8

1.4.3 Evidence-based Approach 8

1.5 Contributions and Dissertation Structure 9

1.5.1 Dissertation Structure 10

2 Trustworthy System Architectures **12**

2.1 System Architecture: Definition and Conceptual Model 13

2.1.1 A System Architecture Captures Runtime Behaviour 13

2.1.2 Conceptual Model of a System Architecture 14

2.1.3 Architecture as a Means of Managing Complexity 16

2.1.4 Architectural Models as Knowledge Representation 17

2.1.5 Capturing the Model Using Formal and Semi-formal Descriptions 18

2.2 Trust in the Context of a System 20

2.2.1 So What is Trust? 20

2.2.2 Conceptual Model of Trust in a System 21

2.3 Trusted Computing Provides Technical Mechanisms for Trustworthy Operation 23

2.3.1 So What are Trustworthy Systems? 24
2.3.2 Trusted Computing as Foundation for Trustworthy Systems . 24
2.3.3 The Trusted Platform Module (TPM) 25
2.3.4 Roots of Trust as Foundations of Trust in a System 26
2.3.5 TPM is Secure Against Software Attacks 27
2.3.6 Secure Storage: Secrets Never Leave the TPM 28
2.3.7 Components are Integrity Measured Before Execution 29
2.3.8 Attestation Enables Trustworthiness Verification 29
2.3.9 TCG Integrity Schemas: A First Step Towards Evidence Collation 30
2.4 Reasoning about Properties of Systems 32
2.4.1 Runtime Architecture Discovery and Verification 32
2.4.2 Reasoning with Security Logic 32
2.4.3 Reasoning about Properties of Trustworthy Systems 34
2.5 Summary 35

3 Architecture-based Framework for Trust-properties 36
3.1 Scenarios 37
3.1.1 Scenario I: Password Manager 37
3.1.2 Scenario II: A Trustworthy Grid Platform 38
3.1.3 Scenario III: Trustworthy Collaborative System 39
3.1.4 Analysing the Scenarios 40
3.2 Framework Overview 42
3.2.1 Objectives 43
3.2.2 Approach 43
3.3 Trust Property Definition 44
3.4 Trust Property Classification 45
3.4.1 Competence-based Properties 45
3.4.2 Outcome-based Properties 47
3.4.3 Composition-based Properties 48
3.4.4 “Shadow of the Future” Properties 49
3.5 Architectural Aspects 50
3.5.1 Structural Aspect 50
3.5.2 Runtime Aspect 51
3.5.3 Interaction Aspect 52
3.6 Mapping Property Classifications to Architectural Aspects 53
3.6.1 Competence- and Composition-based to Structural Aspect . . 54

3.6.2 Outcome-based to Interactions Aspect 55
3.6.3 "Shadow of the Future" to Runtime Aspect 57
3.7 Towards Verification Models 58
3.7.1 Requirements for a Model to Capture Structural Aspects . . . 59
3.7.2 Requirements for a Model to Capture Interaction Aspects . . 59
3.7.3 Requirements for a Model to Capture Runtime Aspects 60
3.8 Summary 61

4 Semantics of Trustworthy Systems 62
4.1 Overview 63
4.1.1 The Need for a Semantic Model 63
4.1.2 Approach and Choice of Model Representation 64
4.2 A Semantic Model 65
4.2.1 Capturing the System Architecture Model 65
4.2.2 Protected Basic System 73
4.2.3 Trust Enablers 80
4.2.4 Trustworthy System Model 89
4.3 Model Validation 91
4.4 Using the Model to Reason about Properties 94
4.4.1 Model Usage Framework 94
4.4.2 Modelling the Password Manager 96
4.5 Evaluation 98
4.5.1 Revisiting the Requirements 98
4.5.2 Lack of Support for Concurrency 99
4.5.3 Making the Basis of Trust Explicit 99
4.5.4 Specification Clarity Versus Provability 100
4.5.5 Comparison with Related Work 100
4.6 Summary 101

5 Abstraction Framework for Reasoning about Interactions 102
5.1 Framework Design Principles 103
5.1.1 Brief Overview of Key CSP Semantics 103
5.1.2 Principle 1: Components as CSP Processes 105
5.1.3 Principle 2: Interactions as Synchronised Events 105
5.1.4 Principle 3: Trust Properties Capture Absence/Presence of Interactions 106
5.1.5 Principle 4: Semantic Models as State Exploration Tool 106

5.1.6 Principle 5: Trust Properties as CSP Specifications 106
5.1.7 Principle 6: Refinement as a Reasoning Mechanism 107
5.2 An Abstract Model 108
5.2.1 Defining a Trustworthy System 109
5.2.2 Model Structure 109
5.2.3 Runtime System 110
5.2.4 Resources 113
5.2.5 Applications 115
5.2.6 System Configurations 116
5.3 Framework Usage 117
5.3.1 System Instantiation 118
5.3.2 Property Specifications 120
5.3.3 Verification Model 120
5.4 Application on a Trusted Grid Architecture 121
5.4.1 Property Specification 121
5.4.2 Common Setup 122
5.4.3 Instantiating a Malicious System 123
5.4.4 Instantiating a Non-Malicious System 123
5.4.5 Property Verification, Results and Discussion 124
5.5 Evaluation 125
5.5.1 Satisfying the Requirements 125
5.5.2 Supporting Virtualised Systems 126
5.5.3 Managing the State Space 126
5.5.4 Comparison with Related Work 127
5.6 Conclusion 127

6 Runtime-based System Trustworthiness 128
6.1 The Challenge of Verifying Trust-properties Using Runtime Aspects . 129
6.2 Trust Properties and Evidence Classification 130
6.2.1 Trust Properties 131
6.2.2 Evidence Classification 131
6.3 Trace-based Verification Model 133
6.3.1 Assumptions 133
6.3.2 Overview 133
6.3.3 Whitelist as a repository of specifications 135
6.3.4 Trace-to-process transformation 136

6.3.5 Property Verification . . . 137
6.3.6 Verifying Virtual Appliances in a Collaborative System . . . 138
6.3.7 Evaluation . . . 141
6.3.8 Summary . . . 143
6.4 Provenance-based Verification Model . . . 144
6.4.1 Graph-based Evidence Representation . . . 145
6.4.2 Reasoning about Trust-properties . . . 153
6.4.3 Verifying Virtual Appliances in a Collaborative System . . . 155
6.4.4 Evaluation . . . 160
6.4.5 Summary . . . 160
6.5 Discussion . . . 161
6.5.1 Satisfying the requirements . . . 161
6.5.2 Reliance on Collection Mechanisms . . . 161
6.5.3 Scalability . . . 162
6.5.4 Generalisation of the Approach . . . 162
6.5.5 Comparison with Related Work . . . 162
6.6 Conclusion . . . 164

7 Application to Trust Domains: Mergers and Acquisitions Scenario 165
7.1 Trust Domains: Enabling Secure Information Sharing . . . 166
7.2 Mergers and Acquisitions Scenario . . . 166
7.2.1 High-level View of an Infrastructure for M&A . . . 168
7.2.2 Use Cases: The Need for Trust Domains . . . 169
7.2.3 Requirements . . . 170
7.3 A Trust Domains-enabling Architecture . . . 171
7.3.1 Components . . . 171
7.3.2 Fitting the Components Together . . . 173
7.3.3 Design Evaluation Process . . . 175
7.4 Applying the Architectural Approach . . . 176
7.4.1 Identifying and Classifying Trust-properties . . . 176
7.4.2 Evidence and Architectural Models . . . 177
7.4.3 Property Verification . . . 180
7.5 Evaluation . . . 182
7.5.1 Benefits for Trust Domains . . . 182
7.5.2 Practical Considerations . . . 183
7.5.3 Challenges . . . 184

7.5.4 Related Work 185
7.6 Conclusions 185

8 Conclusions and Future Work **187**
8.1 Contributions 187
8.2 Evaluation 188
8.2.1 Managing the Size of Evidence 188
8.2.2 Trusting the Collection Mechanisms 189
8.2.3 Impact on Future Designs 189
8.2.4 Privacy Concerns 190
8.3 Future Work 190
8.3.1 Identifying Essential Configurations 190
8.3.2 A Taxonomy for Trust Domains and Trust Indicators 191
8.3.3 A Common Semantics for Expressing Trust-properties 191
8.4 Conclusion 191

A Schema of the Whitelist **193**

B Z Primer **195**
B.1 Schemas as Data Types 195
B.2 Schemas as Predicates 196
B.3 Schemas as Declarations 196
B.4 Schema Operators 196
B.5 Operation Schemas 198

C Complete Semantic Model **200**
C.1 Capturing the System Architecture Model 200
C.1.1 Data and Interfaces 200
C.1.2 Components 200
C.1.3 Connectors 202
C.1.4 Interactions 203
C.1.5 Basic System 204
C.2 Protected Basic System 207
C.2.1 Protection through restricted interactions 207
C.2.2 Protection through memory isolation 208
C.2.3 Protection through ring architecture 212
C.2.4 Combining the Protection Mechanisms 214

C.2.5 Shared Resources 216
C.2.6 Direct Memory Access (DMA) 216
C.2.7 Ring 0 Memory Access 217
C.3 Trust Enablers 217
C.3.1 Integrity Measurement 217
C.3.2 Attestation 219
C.3.3 Trusted Isolation 223
C.3.4 Secure I/O 224
C.3.5 Secure Roots of Trust 224
C.4 Trustworthy System Model 226

D Complete Abstract Model Using CSP **228**

Bibliography **237**

List of Figures

1.1 Illustration of the interfaces that may be provided by a system 3
1.2 Structure of the approach and contributions of each chapter 10

2.1 A model illustrating concepts within a system architecture 14
2.2 Matching the interfaces on the component and connector 15
2.3 Using a model as a way of representing knowledge about a system . . 18
2.4 A core model showing relationships between the Trustor and Trustee 21
2.5 A conceptual model of trust in computer systems. The dotted lines denote a subclass relationship. 22
2.6 TCG Integrity Verification Mechanism 31

3.1 An architecture of a password manager 38
3.2 A researcher using a platform that is part of a grid infrastructure . . 38
3.3 Workflow showing how a virtual appliance is shared in a cloud 40
3.4 Structure of the architecture-based framework 43
3.5 Identification of trust properties classes 45
3.6 An example of a deployment of jobs onto the grid 46
3.7 Identification of architectural aspects (represented by an ellipse) . . . 51
3.8 The mapping between the competence- and composition-based properties to structural aspects 55
3.9 The mapping between the outcome-based properties and interactions aspect 56
3.10 The mapping between the shadow-of-the-future and runtime aspects . 58

4.1 Example illustrating the boot processes of three different platforms . 63
4.2 An illustration of the structure of a component in the model 66
4.3 Constituents of a basic system and operations on them 70
4.4 Constituents of a trustworthy system model and relationship among them 90

4.5 Operations that can be performed on the component and resulting state changes. 93
4.6 Areas of ProZ showing the execution history, state and enabled operations 94

5.1 Interactions are created by overlaying interfaces among the components in the system 105
5.2 A state transition system for the SPEC process 107
5.3 System A sends encrypted passwords to multiple components 108
5.4 System B sends one unencrypted password before sending an encrypted password to S (step 4) 108
5.5 Conceptual view of the model 110
5.6 Structure of the runtime system 112
5.7 Stages in the use of the abstraction framework 117

6.1 Overview of the trustworthiness verification process 134
6.2 An experiment to collect and analyse event logs from virtual appliances 138
6.3 Th main components (Artifacts, agents and processes) of OPM 146
6.4 Using OPM to capture the boot phase 146
6.5 Structure of an extension to support program execution 148
6.6 Structure of an extension to support program configuration 149
6.7 An extension to support capturing integrity measurements using OPM 151
6.8 An experimental setup for generating provenance for virtual appliances 156
6.9 A provenance graph — showing only a small number of entities . . . 158

7.1 An M&A involving the take over of a company, T, by company A . . 167
7.2 High-level view of an infrastructure for M&A 168
7.3 A use case involving configuration management on a system, S_T . . . 169
7.4 A use case involving a regulator performing compliance checks 170
7.5 An architecture to support trust domains for M&A 174

A.1 Whitelist schema — an extension of *RimmType* from TCG RM Schema 193

Chapter 1

Introduction

Computer systems play an increasingly significant role in almost all aspects of our lives, including education, entertainment, communication and business. Critical services such as life-support, water treatment and power grids — in which faults or failure could have catastrophic consequences — have also seen an increased reliance on computer systems [83, 28]. This reliance has led to a proliferation of sensitive data handled by such systems. Consequently, computer systems have not only become valuable to their owners and users but have also become a rich target for attackers wanting to exploit them for financial gain, political reason or for fun. As a result, the need to build always-available, accurate, confidentiality- and integrity-preserving systems that continue to operate under malice, error or mischance [9], has never been greater.

This challenge can be recognised as a problem of enabling trustworthy operation. In a memo to Microsoft employees, Bill Gates [55] discussed the importance of enabling trustworthy operation of computer systems in business — emphasising the need to better manage networks and to demonstrate to clients that critical business systems are safe from harm. Towards this end, and based on the realisation that "software alone cannot protect itself" [25], *trusted computing* has been proposed to provide hardware-enhanced mechanisms that enable cryptographic identification of computer systems and their configurations in such a way that this identity cannot be falsified even in the presence of malicious code.

However, establishing that a system is trustworthy is still a challenge that not only continues to receive considerable attention but also hinders adoption of systems such as those based on grid or cloud computing [125, 87]. Towards this end, several systems, including grid computing [30, 94, 155], cloud computing [149], network access control [164], log reconciliation [71], mobile computing [52] and storage [164], propose the use of trusted computing as a means enabling trustworthy operation.

Though these architectures are all dubbed "trusted" or "trustworthy", not much attention has been directed towards providing a systematic justification for the basis of this trust — in a way that is both meaningful to other entities and amenable to automated and/or formal reasoning with respect to properties expected by relying parties. Consequently, it is still not clear: i) how the use of such mechanisms influence the overall trust in a system; ii) the properties and assumptions upon which trust is based; and iii) the nature of evidence necessary to reason about these properties [173]. These problems can be attributed to a number of factors including: i) the complexity of modern systems — brought about by increasing feature sets and the need to operate in dynamic environments; ii) the lack of consensus on a definition of trust; and iii) the lack of a systematic approach for using such evidence to understand trust-related properties of a system.

The process of architecture design relies on the use of abstraction mechanisms to simplify design descriptions as well as interactions with the system. A system architect would create graphical representation of a system and present that to stakeholders such as project managers. Similarly, a set of application programming interfaces (APIs) could be defined as part of the system to provide a (simplified) standard interface for interacting with the system. These abstractions are used throughout the design and implementation phases of a system.

In this dissertation, we leverage these abstractions to specify properties and represent evidence from a system so as to enable reasoning about such properties. We argue that an architectural approach towards trust establishment provides suitable abstractions that enable the properties and assumptions, upon which trust in a system is based, to be made explicit. We, therefore, address these challenges by proposing the use of architectural models as a means of representing evidence about a system's behaviour and to reason about properties of interest to the entity wishing to rely on the system.

We develop a framework that identifies and categorises trust-related properties and maps them to architectural aspects that could be used to capture evidence necessary to reason about such properties. Guided by this framework, we develop architectural models that are suitable for capturing these aspects and for reasoning about the properties identified in each category. These models include: i) a semantic model that characterises the building blocks of trustworthy systems in order to demonstrate how trustworthiness develops from the constituents of the system; ii) a CSP-based abstraction framework for capturing the interactions within a system and reasoning

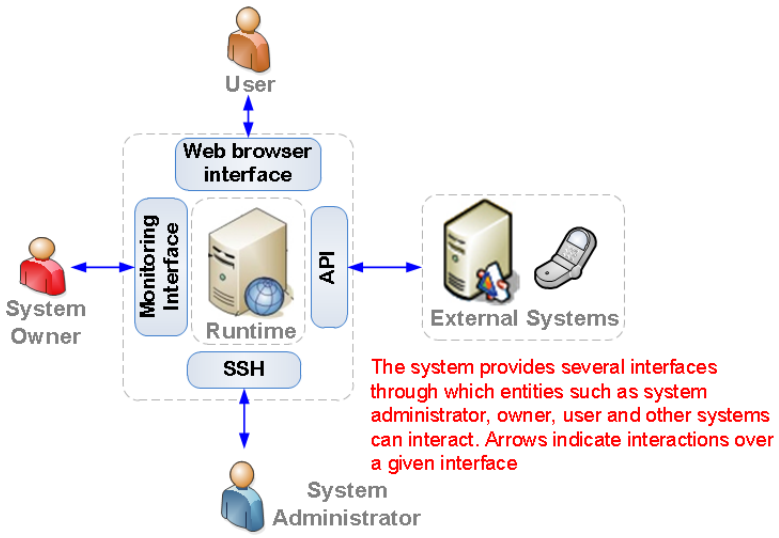


Figure 1.1: Illustration of the interfaces that may be provided by a system

about the kinds of behaviour that can result from these interactions; and iii) a trace-based mechanism that enables analysis of runtime evidence in the form of event logs and a more general provenance-based mechanism in which evidence is collected in the form of a provenance graph with sufficient semantics to capture and allow verification of trust-related properties.

This approach can be employed during the design phase of a system architecture, and to a limited extent during its operation. The approach relies on being able to identify the evidence and choosing one, or a combination, of the proposed models to represent a system. By using this approach system architects can better understand the properties and assumptions that might influence how trustworthy a system would be for a given purpose.

1.1 Defining Trustworthy Systems

This dissertation focuses on trustworthy systems. But what are trustworthy systems? To answer this question, we begin by noting that every system is designed to fulfil certain goals such as provision of a service, storage of data or execution of tasks [30, 94]. Furthermore, any useful system will interact with other entities such as systems within a larger system or humans, such that each entity may rely on some aspect of the system in order to achieve certain goals.

As an example, consider the *webinos* system [174] — a cross device platform for web applications — as illustrated in Figure 1.1. The system may be relied on

by various entities (in this case, four entities including *User*, *System Administrator*, *System Owner* and *External Systems*). The *System Owner* wants to ensure that the activities of the other entities interacting with their system does not negatively affect the operation of the system; the *System Administrator* wants to ensure that the system observes the policies she sets; the *User* seeks to ensure that the system does not leak the sensitive data she passes to it; and *External Systems* such as other devices expect the system to correctly perform computations and return the correct results.

Of particular interest in the context of this dissertation is the ability of such a system to demonstrate to relying entities that it operates as expected. This can be achieved by using a system architecture with intrinsic support for enabling other entities to establish trust in the system. We call these systems trustworthy systems and define them according to the TCG notion of trust [128] and the RFC 4949 [156] definition of trustworthy system as:

> ***a system that always behaves according to the expected manner for the intended purpose and allows relying entities to verify its behaviour against their expectations.***

Verifying that such a system is indeed operating as expected is, however, a challenge especially considering that several abstractions may be provided to relying entities to simplify interactions with the system.

We argue that these abstractions can be used to reason about trust properties, defined in Section 3.3, by following an architectural approach, where the behaviour is determined by collecting evidence and representing it in a format suitable for reasoning about trust-related properties. These properties capture a particular entity's expectations and thus verifying them against the evidence makes the properties and assumptions, upon which trust is based, explicit.

1.2 The Need to Reason about Trust Properties

The need for trustworthy systems has been discussed in many fora. In 2000, Schell and Thompson [150] stated that platform architectures lacked foundations for protecting against attacks and suggested the use of additional hardware and software components to improve this situation. Bill Gates' memo [55] to Microsoft employees emphasised the need to demonstrate how critical business systems were protected from malicious

activities. The Trusted Computing Group (TCG)[1] has developed technologies to enable trustworthy operation. But why do we need to reason about trust-related properties? In the rest of this section, we discuss why this may be necessary.

1.2.1 The Inexplicable Basis of Trust

One approach towards enabling trustworthy operation involves the use of trusted computing mechanisms such as integrity measurement, remote attestation and secure storage. Using these mechanisms, a system's configurations are securely collected and reported to a relying entity to enable that entity to make a decision about whether or not the system in question is configured appropriately. However, not much attention has been directed towards understanding the nature of such information or how it can be used to come to such a conclusion. The focus has been on finding ways of employing the proposed technical mechanisms to enhance trustworthy operation. As a result, little is known about how such mechanisms contribute to the overall behaviour of a system. In other words, given that certain technical mechanisms are in place, how can we come to a conclusion that the system employing such mechanisms will be trustworthy? This concern has been raised at many fora such as the European Trusted Infrastructure Summer Schools (ETISS)[2].

1.2.2 Increased Complexity

The level of complexity in computer systems has continued to increase as more functionality is demanded of such systems. If such systems are to be used for critical operations or to store or operate on critical data, it is important that their possible behaviours are considered to ensure that they conform to certain expectations. To manage complexity, a number of abstraction layers are built on top of each other such that each layer encapsulates the finer details of the underlying layer and only exposes information that is necessary for a particular need. But how can we take advantage of simplifications enabled by abstractions to reason about a system's ability to conform to the expected operation?

[1]The TCG is a not-for-profit organisation formed to develop, define and promote open, vendor-neutral, global industry standards, supportive of a hardware-based root of trust, for interoperable trusted computing platforms.

[2]The European Trusted Infrastructure Summer Schools are organised and attended by researchers and industry experts in trusted computing and system security. The event covers presentations on both academic and industry research, practical sessions and panel discussions.

1.2.3 System Dynamics

Computer systems are designed with multiple configuration options, which may affect their operation [59]. A distributed system, for example, can be configured to support load balancing — sharing requests for the system's services among a number of resources — or dynamic scalability — where the capacity of the system is increased or decreased in response to demand for its services. These configurations introduce dynamism in the structure and state of the system, which may lead to unexpected behaviour [2]. This brings up the question: how can we guarantee that the system will still operate in accordance with certain expectations given that its state continues to change? Coming up with an answer to this question would require understanding how each possible configuration could affect the behaviour of the system as well as a way of determining when the system can assume a particular configuration.

1.2.4 Attackers are Opportunistic

Despite continued efforts in security and trustworthy system design, the number and level of sophistication of attacks continue to increase, as demonstrated by the growing catalogues of system attacks [167] and vulnerabilities [168]. One reason for this trend may be that attackers care less about the defence mechanisms in place. Instead, they are always seeking for opportunities arising from mis-configurations, design flaws and errors in the operation of the system, to achieve their goals. How can we determine that a system is not configured to mount attacks on other systems [140] or to be used by malicious entities as a distribution point of further attacks (i.e. as part of a bot net)? To determine a system's capability to defend itself against attacks and to protect its computation and data, we need to understand how its configurations interact with its construction and be able to reason about what kinds of behaviour result from a particular combination of the two.

1.2.5 Multiple Entities Rely on the System

As discussed in Section 1.1, and illustrated in Figure 1.1, in any given system, there will be multiple entities such as humans and other systems that will rely on the system for one or more of its services. By the TCG's definition of trust, each entity will have concerns that reflect the purpose for which it intends to use the system.

Furthermore, the use of abstraction mechanisms to simplify interactions with the system imply that different information (both in terms of amount and level of detail) will be available to each entity. This means that no single entity will have a complete

view of the system. Instead, each entity will have a perspective of the system that is determined by the amount of information it has and will, therefore, have to establish trust in a system based on this perspective.

1.3 Thesis Statement

State-of-the-art in trustworthy system design concentrates on techniques for enabling trustworthy operation. However, less attention has been directed towards understanding how these techniques result in trustworthy operation. This has a danger that trust may be misplaced and that we may continue building more of such systems without a clear understanding of the basis for the trust placed in such systems and the assumptions upon which it based. In this dissertation, we make a case that:

> ***an architectural approach provides effective abstractions for making trust properties and assumptions explicit and reasoning about a system's ability to satisfy these properties***

This thesis has been influenced by a number of factors including observations of current research in trusted computing and future trends, building on previous work [118], and discussions with leading experts in the field. One issue that has come up in several fora has been how to generate trust-related proofs about a system and how to verify these [173]. For this reason, this dissertation makes a contribution by proposing the use of an architectural approach as a way of reducing the complexity associated with establishing trust in a system. We provide definitions of key terms and concepts used in the dissertation in Table 1.1.

1.4 Overview of the Approach

The overall approach adopted in this dissertation is based on a combination of three approaches: *architectural*, *building block* and *evidence-based*.

1.4.1 Architectural Approach

Software architecture is an essential means of dealing with complexity in computer systems. A software architect produces blue-prints of the system and uses them to communicate to various stakeholders. This approach enables the architect to only include details in a particular view of the system necessary to convey certain information to a particular group of stakeholders.

Term	Description
Trust	Is the expectation that a system always behaves in the particular manner for the intended purpose.
Evidence	Information that can be used to support trust assertions.
Architecture	An embodiment of elements and relationships among them, which leads a system to exhibit certain runtime behaviours.
Trust property	An assertion about actions and/or results expected from a system or environment in which it runs, that contribute to a consistent behaviour of the system.
Trustworthiness	The extent to which an entity satisfies the trust-properties deemed essential for a particular purpose.
Trustworthy System	A system that always behaves according to the expected manner for the intended purpose and allows relying entities to verify its behaviour against their expectations.
Provenance	A description of how an entity came to be in a particular state.
Reasoning	The process of determining whether or not, or to what extent an assertion can be satisfied given certain facts.

Table 1.1: Definitions of key terms and concepts used in the dissertation

We use an architectural approach in two ways. Firstly, we create a conceptual model of system architecture and use it to identify aspects of a system that could be used to reason about a given category of trust-related properties. Secondly, we create architectural models that can be used to represent information from each aspect in a manner suitable for verification against trust-related properties.

1.4.2 Building Block Approach

This approach is inspired by Grawrock's [59] idea of viewing a trustworthy system as consisting of inter-dependent elements with functional roles within the system. We refer to these elements as *building blocks* because they form the foundations of the system's capabilities and are used to capture the structure and composition of a system. In this approach, we make use of formal specifications to represent a system architecture and reason about its ability to satisfy competence and composition related properties.

1.4.3 Evidence-based Approach

During the operation of a system, several remnants may be produced in various forms including data consumption, data generation and connection creations. These remnants could be used to build an understanding of the behaviour of the system [139] and could, therefore, be treated as evidence about a system's behaviour. This dissertation adopts an evidence-based approach to represent information about a

system's trust assertions and to reason about whether or not the assertions represent trustworthy operation.

This approach is achieved through a combination of two techniques: formal trace analysis and provenance [116]. A formal trace analysis that uses Communication Sequential Processes (CSP) [68] to represent the activities on a system in terms of communicating process models is developed. We make use of provenance models [63] to represent the evidence as a graph and then verify properties by performing queries on the graph or sub-graphs that satisfy certain properties.

1.5 Contributions and Dissertation Structure

The dissertation builds on a number of ideas from research in distributed systems and security and uses several theories and approaches to identify the problems and develop solutions for them. In so doing, the thesis makes the following contributions.

- *An architecture-based trust framework* — identifies the link between trust and system architecture and provides a means of matching properties expected of systems to its architectural aspects.

- *A semantic model to demonstrate how trustworthiness is influenced by the building blocks* — we argue that the concepts used and the relationships among them provide one perspective from which the trustworthiness of a system can be understood. We capture the concepts in a trustworthy system using a model based on the semantics of *Z* [159] and use this model to reason about the basis of trust in these systems.

- *Trustworthiness via architectural abstractions* — to understand the influence of interactions on the trustworthiness of a system, we develop a model based on CSP [68] that enables instantiation of systems with varying trust properties, i.e. depending on how the system is constructed and configured. We further develop a framework that guides the usage of the model, enables compositions of systems to create more complex systems and verification of resulting instances.

- *Trace-based verification for model* — we have developed a mechanism that is based on trace analysis to enable analysis of runtime event logs in order to verify the trustworthiness of a system. Event logs are collected, filtered and transformed into a CSP process that is then verified against a whitelist, which is captured as a repository of specifications.

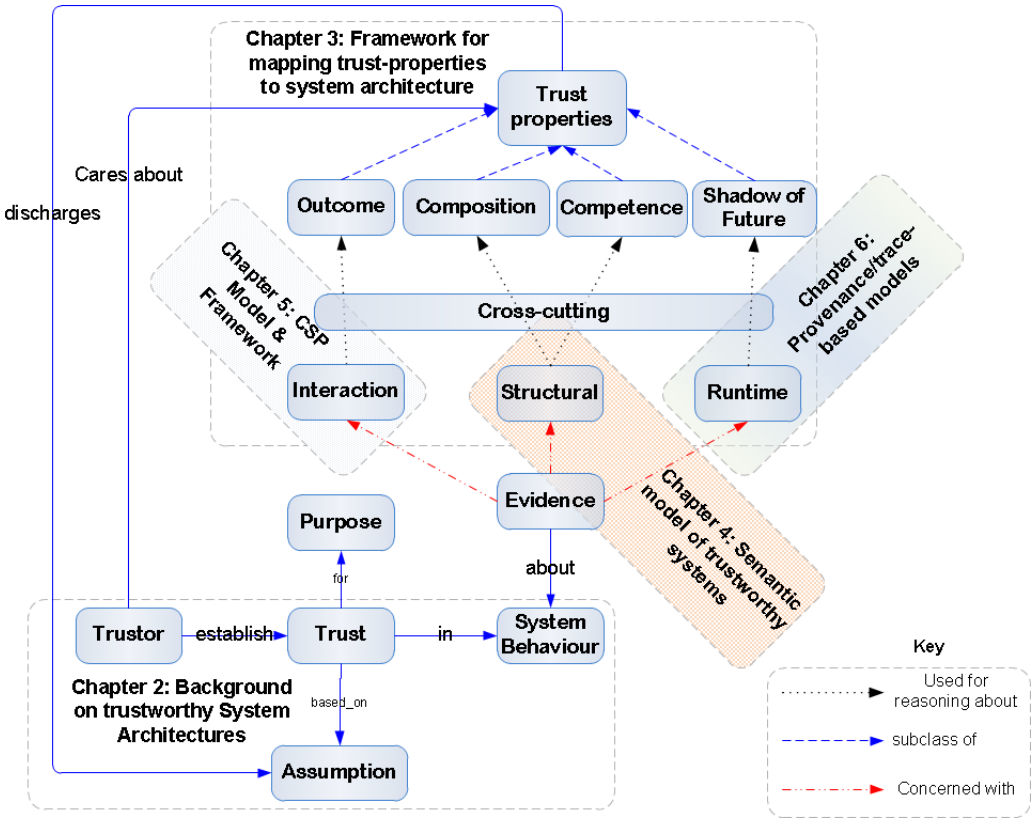


Figure 1.2: Structure of the approach and contributions of each chapter

- *Provenance-based evidence representation and verification* — we demonstrate that provenance is a rich source of runtime evidence that can support verification of trust-related properties. We, therefore, extend existing provenance models and develop a graph-based representation that captures evidence in a manner suitable for trustworthiness verification.

These contributions have been presented in a number of publications including *an abstract model for trusted platforms* [120], *trustworthiness verification in distributed systems* [119] and *provenance-based model for trustworthiness verification* [121].

1.5.1 Dissertation Structure

The structure of the dissertation is illustrated in Figure 1.2. Chapter 2 investigates the literature and presents the theoretical foundations upon which the proposals in this dissertation are built. These investigations also guide us in designing solutions for the problems identified in this dissertation.

Chapter 3 introduces a number of scenarios that demonstrate the importance of evaluating the trustworthiness of a system. These scenarios form the basis of the

investigations in the rest of the dissertation and are used to identify and classify trust-related properties. This chapter also identifies a mapping between classes of trust properties and aspects of a system that may be used to reason about such properties and based on this mapping identifies key requirements for models that can be used to capture knowledge about a specific aspect.

Our first proposal involves the use of a semantic approach to characterise the concepts within trustworthy systems and formulates mechanisms for reasoning about emergent properties for systems that use these concepts. This approach is presented in Chapter 4 and includes a semantic model, which captures concepts and relationships among them as well as properties induced by their inclusion in a system architecture and assumption upon which their use relies.

Chapter 5 takes the view that a system's trustworthiness depends on its *construction* and *configuration*. The chapter, therefore, goes on to describe an *abstraction model* that can be used to capture the construction and configuration of a trustworthy system. The second part of the chapter describes a framework that guides the use of the model as well as verification of the properties of resulting instances. Finally, the instantiation and verification of a trustworthy grid platform is discussed.

Chapter 6 concentrates on the runtime aspects of the system. In this chapter, a verification model is proposed that is based on transforming event logs into a process model that is then verified against specifications. In addition, the chapter describes a second proposal, which is more generic, for capturing runtime aspects using provenance.

Chapter 7 demonstrates the applicability of the ideas developed in this dissertation to trust domains. A scenario involving a system that is used in mergers and acquisitions transactions is used to motivate the need for trust domains and an architecture for enabling trust domains is developed. We then demonstrate how the ideas proposed in this dissertation can be integrated into trust domains and how they can serve as a foundation for enabling relying parties to establish the properties of a trust domain.

We summarise our findings and conclusions in Chapter 8. This chapter also discusses the gaps in the thesis and makes suggestions about how these could be addressed.

Chapter 2

Trustworthy System Architectures

In this chapter we describe the background necessary to explore the problems that will be addressed in this dissertation and develop solutions for them. We begin by defining a system architecture and presenting a model that includes the fundamental concepts, i.e. components, connectors and data, in a system architecture and how these concepts are related. This model is used as a foundation, in Section 3.5, for identifying aspects of an architecture that can be used to reason about various classes of *trust-properties*. We then investigate, in Section 2.1.5, existing mechanisms for representing a system architecture and their shortfalls with respect to reasoning about trust-properties.

The notion of trust is used in several places in this dissertation and, therefore, deserves some attention. For this reason, in Section 2.2 we investigate various definitions, frameworks and models of trust in order to establish what such a notion implies when used in the context of a system. Based on this investigation, we develop a conceptual model that includes concepts used to define trust and how these are related when used in the context of a system. This model is used to identify classes of trust-properties in Section 3.4.

In Section 2.3 we investigate how techniques introduced by *trusted computing* enable trustworthy system operation. This investigation reveals that although these techniques continue to be used in the design of several systems, it is still not clear how their use influence trust-related properties. Furthermore, current state-of-the-art pays less attention towards exploring this issue, instead attention is mainly directed towards the use of these techniques. We identify some notable contributions towards addressing these issues in Section 2.4.

2.1 System Architecture: Definition and Conceptual Model

Most non-trivial computer systems comprise of multiple components — possibly, distributed across multiple virtual and physical spaces (as is the case for distributed systems such as those based on cloud computing [134, 124]) — interconnected by complex communication infrastructure. Such systems are also expected to dynamically adapt to changes in the environment such as increased workload or failure of some components. This results in systems which are complex making it challenging for any single stakeholder to understand or predict the behaviour of a system.

Complexity is generally managed through abstraction mechanisms [182, 114, 66] in which details that are less important for a particular aspect of a system are ignored, allowing the relying party to concentrate on more important details. Such abstractions can be achieved by taking an architectural approach to system design, in which a system architecture is used to capture the structure and behaviour of the system while enabling multiple perspectives that may be useful for understanding certain aspects of a system. This approach necessitates a definition of system architecture, which we provide in the next subsection. Based on this definition, we develop a characterisation of the concepts that exist within a system architecture. We capture these in a model which we later use in Section 3.5 to identify architectural aspects and in Section 4.2 as a foundation for a semantic model.

2.1.1 A System Architecture Captures Runtime Behaviour

The IEEE 1471-2000 standard [76] defines a system architecture as "the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." This view of architecture in terms of components is commonly adopted in software architecture research [130, 54, 50]. Using insights from civil architecture Perry and Wolf [130] define an architecture as a set of architectural elements (processing, data or connecting elements) that have a particular form. Garlan and Shaw [54] further specify that such a set must be accompanied by a description of the interactions between them. Fielding [44] argues that an architecture should not only capture the structure of the system but, most importantly, it should capture the runtime behaviour of the system. He, therefore, defines an architecture as a "runtime abstraction of a system".

We build on these definitions and define an architecture as:

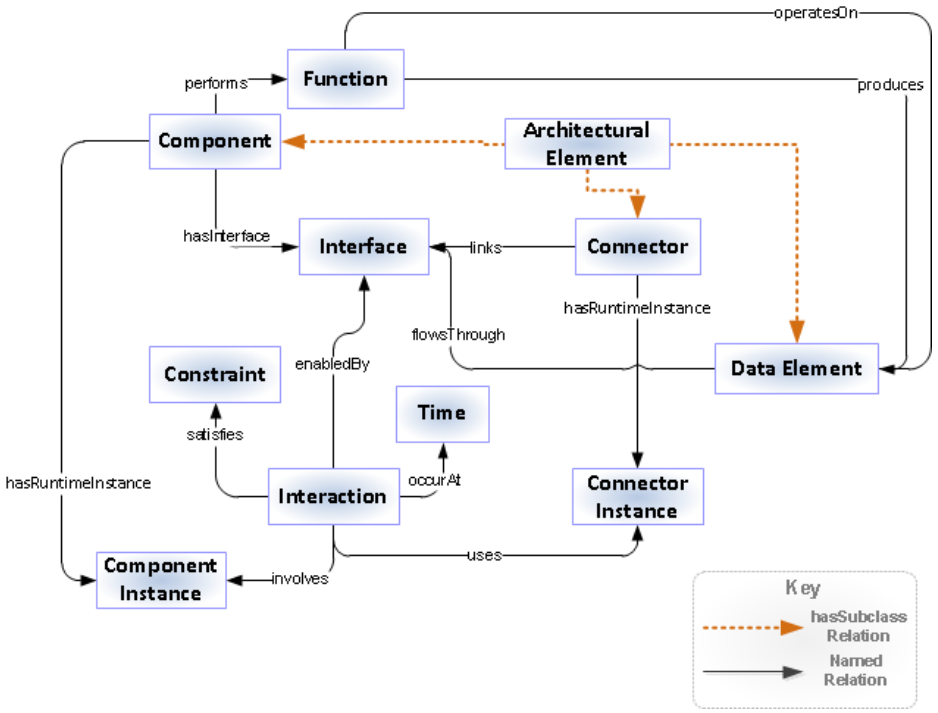


Figure 2.1: A model illustrating concepts within a system architecture

> ***an embodiment of elements and relationships among them, which leads a system to exhibit certain runtime behaviours.***

In the next section we develop a model that defines concepts used to capture a system architecture and how these concepts are related.

2.1.2 Conceptual Model of a System Architecture

An architecture comprises of a number of *architectural elements* [44, 130], or simply elements, that serve as fundamental pieces from which a system is constructed. These exist in three main forms: i) *component*, ii) *connector*, or iii) *data*. We capture these elements together with relationships among them in a model illustrated in Figure 2.1.

Components as Processing Units

As discussed in Section 1.1, a system provides one or more services to other entities, such as users or other systems. In order to achieve this, a system has to include elements that are capable of performing computations and provide means of transporting the results of the computations either within the system or to external entities. This is achieved by architectural elements called *components*.

Components, also known as processing elements [6], perform computations which transform data [130]. These could exist as hardware, firmware or software. For example, a grid computing system might include components such as web servers, resource brokers, schedulers and policy enforcement points. These communicate among themselves or with the environment. To achieve this, each component provides one or more interfaces [113] through which data can flow into or out of the component. Each interface could consist of multiple channels or ports [141] that have different roles. For simplicity, we restrict the definition to interfaces rather than go as far as ports.

Components may reside within the same physical or virtual space — as is the case of a single platform, or spread out in multiple, possibly geographically dispersed locations — as in the case for distributed systems [30, 94, 149, 134, 124]. This necessitates interaction mechanisms such as networks, shared memory, remote call procedures and web services, which may introduce another layer of complexity in a system. These mechanisms are achieved through the notion of *connectors*.

Connectors Enable Interactions

Connectors enable communication among components in a system and thus provide mechanisms for moving data within the system or to other systems [6]. As a result, they serve as "glue" that hold the elements together [8, 130].

Given a connector C with interfaces i and j, and components A and B, then A and B can only interact if there exist some interface k at A and l at B such that each of k and l is compatible to at least one of i or j. Figure 2.2 illustrates how a connector with two interfaces can fit into an interface from component A and another interface of component B. For example, the *http* channel can be defined as a connector that can allow web browsers as well as terminals (such as *bash* or Windows CMD) to connect to a web server.

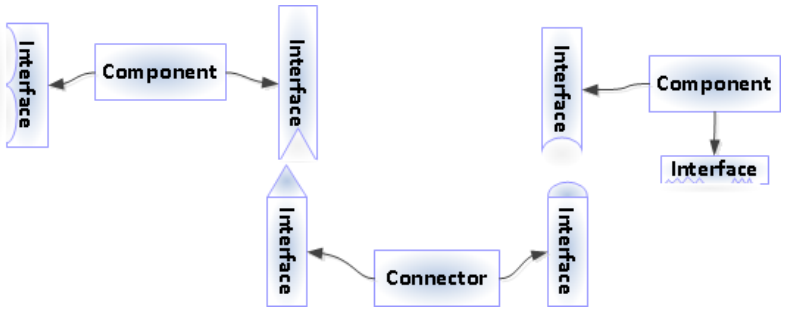


Figure 2.2: Matching the interfaces on the component and connector

Data as the Essence of Computing

In addition to components and connectors, an architecture also includes data [6, 130] that can be transformed by computational components. Data serves as input to and output from functions that are performed by the components and occur in many forms including policies, credentials, and configurations.

When a system is running, it accepts data through its input interfaces, such as keyboard, touch screens or network, interpret the data, transform it or use it to make decisions. Data is also used in controlling how other elements should behave. For example, configuration files are used to specify operational parameters for components on a system.

In certain cases, there may not be a clear distinction between data and other elements. For example, *bash* scripts on Unix systems can be used to perform certain functions and, therefore, qualify as components. But they can also be used as input to other components, such as text editors and can, therefore, be regarded as data.

Runtime Artifacts

We earlier defined an architecture as a *runtime abstraction*. In order to capture this notion, we need to identify architectural concepts that exist when the system is instantiated. The architectural elements discussed above can be thought of as design artifacts that require instantiation when the system begins to run [141]. For this reason, we define *component instance* and *connector instance* as runtime artifacts of the architectural elements *component* and *connector*, respectively. These instances must occur within the same time space in order to create interactions.

We define an *interaction* as an activity that involves two or more component instances that use the same connector instance to communicate. An interaction also specifies the interfaces that are used on both sides of the communication, the *time* at which it occurs and the *constraints* that have to be satisfied by the component and connector instance in order for it occur. These constraints may be specified in terms of the types of components allowed or the state they should be in.

2.1.3 Architecture as a Means of Managing Complexity

The elements discussed in the previous section are combined to provide services for which the system is designed and in a manner that induces some desirable properties

[44] such as reliability or scalability. This combination might lead to system complexity — as each element might be a complex sub-system in itself communicating with other sub-systems over complex infrastructure.

Complexity presents a challenge both in the design and the analysis of systems. In the context of security and trust, it is important that mechanisms for managing this complexity are in place. In this regard, abstraction mechanisms [182, 114, 66, 101] — in which less important details are ignored while concentrating on only those that are essential for a particular aspect of interest — may be employed. Such mechanisms could be provided by an architectural approach in that a system architecture could be used to provide a skeleton for understanding system properties at different levels of abstraction [154].

An architecture enables complex systems to be decomposed into more manageable subsystems which are easier to understand. However, in most cases it may still be a challenge for every stakeholder in the system to understand all aspects of the architecture. For this reason, an architecture may be represented in terms of architectural views, defined in the IEEE 1471-2000 standard [76] as "a representation of a whole system from the perspective of a related set of concerns". So that, rather than having a single representation that captures the concerns of all stakeholders, multiple concurrent views can be used, allowing each view to address concerns of various stakeholders [91].

2.1.4 Architectural Models as Knowledge Representation

Views are composed of models, where each model is an abstraction of some knowledge about a system [6]. This knowledge describes how the system provides the services it is designed for and the circumstances under which they do so. Models also specify the activities carried out by an operational system including the processes performed or programs invoked, information flows and data used or created.

Figure 2.3 illustrates how models relate to knowledge about the system and stakeholder concerns. In this dissertation, we make use of views to decompose the problem of determining how trustworthy a system is so that each view captures some aspect of the system that can be used to reason about certain kinds of trust-related properties. We then develop architectural models to capture knowledge about some identified aspects that might be useful for reasoning about trust-related properties.

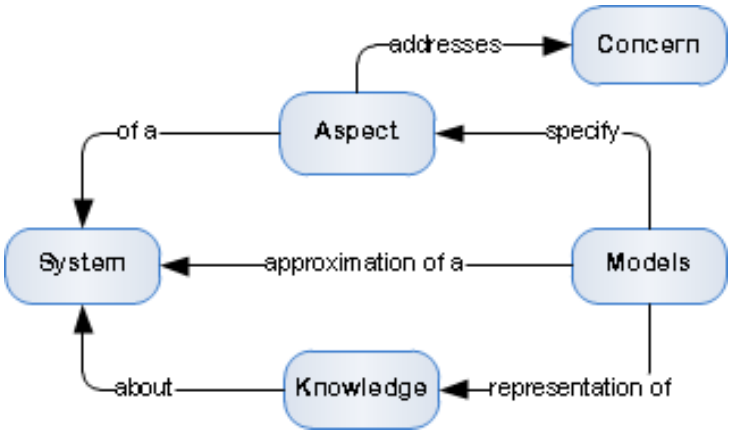


Figure 2.3: Using a model as a way of representing knowledge about a system

2.1.5 Capturing the Model Using Formal and Semi-formal Descriptions

An architecture facilitates communication among stakeholders and analysis of system properties by using views and models as discussed above. To achieve this, the models discussed in the previous section must be captured in a manner suitable for both formal and semi-formal analysis. This is provided by most architecture description languages (ADLS) [111]. Research in ADLs is very active, with numerous proposals and, therefore, a comprehensive study of ADLs could take up an entire chapter. Instead we focus on those that may potentially be useful for investigating the kinds of problems we are interested in.

The Unified Modeling Language (UML) is a well known standard for describing the structure and behaviour of a system. It provides class diagrams which can be used to model relationships among objects in a system and statecharts for analysing the behaviour of the system. Enhancements in UML 2.0 make it particularly useful for architectural descriptions. For example, the concept of ports [10] can be used to capture interactions among system components. However, UML has been noted to lack semantics essential for formal reasoning [132, 145]. Pradella et al. [132] propose ArchiTRIO as a temporal logic based language that complements UML 2.0 to enable formalisation of the structure and behaviour of the system. Similarly, Rossi et al. [145] propose the use of temporal logic to formalise the semantics of UML state machines such as transitions and behaviours.

The software architecture model (SAM) [37] is a framework for modelling and analysing software architectures. It is based on two complementary formalisms: Petri-nets and temporal logic. In SAM, a model is represented as a set of elements

(components and connectors) that have behaviour, B, and a set of properties, S. The behaviour is captured using a high-level form of Petri-nets and properties are specified using temporal logic. The elements are then grouped together to form a composition $C_i = \{S_{ij}, B_{ij}\}$, which is analysed by verifying that every B_{ij} satisfies S_{ij} using techniques such as temporal logic proofs, reachability tree [70] and structural induction [92].

xADL is an XML-based architecture description language that supports architectural constructs including components, connectors and interfaces [89]. An extension to this language, secure xADL [138], has further been developed to enhance the representation of security related aspects of an architecture.

The Trustworthy Architecture Description Language (TADL) [113] is designed to describe trustworthy component-based architectures enabling verification of trustworthiness properties, defined in terms of availability, reliability, safety, and security, during the design stage. In TADL, security is considered to be a design-time property. Consequently, the language does not provide means of capturing the runtime security state of a system.

WRIGHT [141] is a formal architecture description language based on a variant of Communicating Sequential Processes (CSP) [144, 68]. Each component in WRIGHT is described in terms of the interfaces to its environment and the computations it can perform. Interfaces capture some aspect of the component behaviour while the computations tie the ports together to describe the overall system. Connectors are described in terms of the "roles" and "glue". Roles indicate the behaviour of a single participant in an interaction while the glue specifies how the participants work together to create an interaction. WRIGHT also defines the concept of a configuration as a particular arrangement of components using specific connectors. This is captured using instances of components and connectors and their attachments, where an attachment is made by pairing a components port to a connector's role. Our work borrows a number of concepts from WRIGHT and modifies them to make them useful both for describing trustworthy systems and reasoning about trust-related properties. For example, the concept of a port in WRIGHT is equivalent to an interface in our case. We also do not distinguish between ports and roles, instead we simplify the connection between components and connectors to be facilitated by interfaces that exist on both elements.

2.2 Trust in the Context of a System

In the previous section we discussed the issue of complexity in computer systems and how this can be managed by following an architectural approach. In this section we investigate what it means to "trust" such systems. The main challenge is that the notion of trust can be used to mean different things in different context. In spite of significant effort invested in research on various aspects of trust, there is still little consensus on its definition [67, 180, 90] and others [123, 139] argue that such a notion cannot be applied to non-cognitive entities. Therefore, applying such a notion to computer systems, which are themselves complex, leads to challenges not only in the conceptualisation of what it means for a system to be trusted but also in establishing this fact. We address this challenge by defining what trust means in the context of a system and capturing this notion in a model.

2.2.1 So What is Trust?

Trust is commonly specified as a relationship in which one entity, a *trustor*, relies on another entity, referred to as a *trustee*, for a particular "*thing*". Such a "thing" could take many forms, such as provision of a service [58], access to a resource owned by the trustor [58] or an action that can be performed by the trustee [84]. It is normally expected that such a "thing" could be at stake [139] or have an impact on the trustor's welfare [84].

The trustor and trustee could be of many forms, including humans, systems, organisations, group of systems and so on. However, in this dissertation, we are interested in scenarios where the trustee is a system; more specifically one built on trusted computing primitives. Therefore, our notion of trust is based on that proposed by the TCG [128], so that:

> ***a system is trusted if it always behaves in the expected manner for the intended purpose.***

This means that expectations are specified from the point of view of the trustor, and in relation to the purpose for which they rely on the system, so that a given system may be deemed trusted by one trustor but not trusted by another entity.

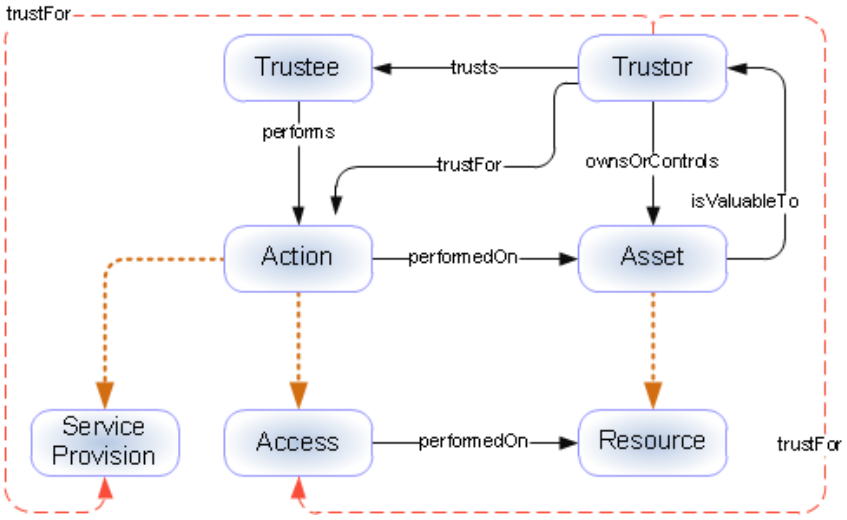


Figure 2.4: A core model showing relationships between the Trustor and Trustee

2.2.2 Conceptual Model of Trust in a System

We develop a conceptual model of trust that is based on the definition provided in the previous section. The model illustrated in Figures 2.4 and 2.5 captures the relationship between a trustor and a trustee in terms of the actions expected of the trustee. This model helps us in identifying and classifying properties that could be used to specify expectations of behaviour.

The *trusts* relationships is captured as a one directional relation from the *Trustor* to the *Trustee*. We use the relation *performs* to denote the relationship between the *Trustee* and *Action* and *performedOn* to capture the idea that an *Action* could be performed on an *Asset*, where an *Asset* is defined as something owned or controlled by and valuable to the *Trustor*. For example, in a grid environment, a researcher (a trustor) could trust a grid platform assigned to them to execute a job (an action) on the data for the experiment (asset) [31].

When using the *trusts* relationship, it is not enough to simply say "an entity E *trusts* the system S". Instead, the statement has to be qualified to specify the kind of thing for which S is being trusted so that one could say "E trusts the system S *for* the execution of the algorithm". To capture this, we define *trustFor*, based on the conceptualisation by Jøsang et al. [84], as a relationship that identifies the kind of "thing" for which the *Trustee* is being relied on. It relates the *Trustor* to the *Action* expected of the *Trustee*. Furthermore, both access to a resource and service provision (identified by Grandison and Sloman [58] as "the things for which the trustee could be trusted for") can be considered as actions that can be performed by the *Trustee*.

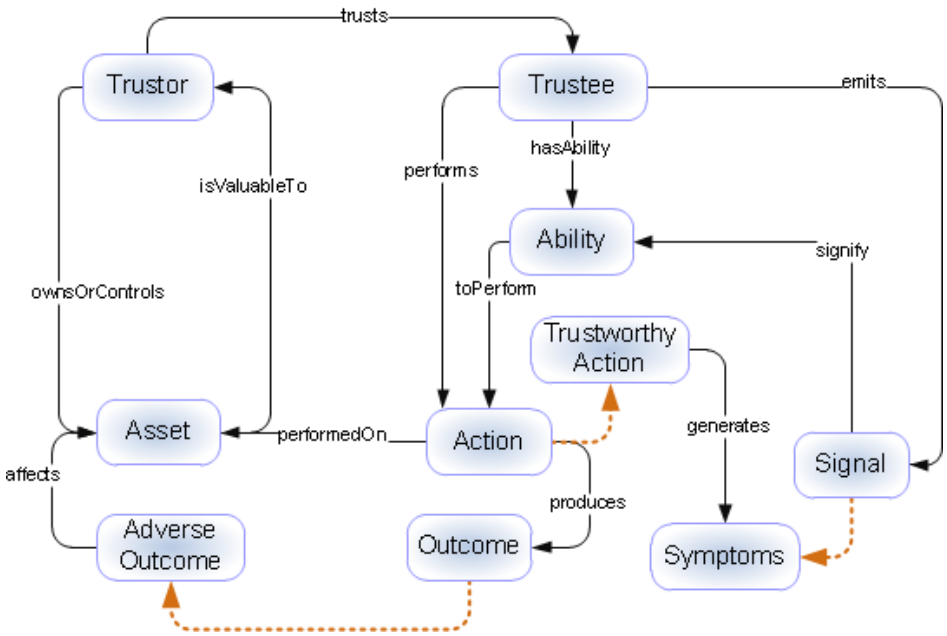


Figure 2.5: A conceptual model of trust in computer systems. The dotted lines denote a subclass relationship.

We can infer that the *trustFor* relationship also links the *Trustor* to *Access* and *ServiceProvision* actions (these are indicated by dashed lines).

So far, the model captures trust relationships in terms of actions that the trustee performs. However, it does not capture the idea that an *Asset* should be something at stake. This is achieved by looking at what happens when an action is performed. Riegelsberger et al. [139] note that there has to be a possibility of adverse outcome from actions that can be performed on an asset. Similarly, Xiu and Liu [179] note that when actions are performed, some *effects*, such as state change or message exchange, are produced. We, therefore, define *produces* as a relationship between an *Action* and an *Outcome* to capture the idea that every time an action is performed, some outcome is produced. *Adverse Outcome* is defined as a subclass of *Outcome* to capture the idea that not all outcomes from a particular action may be desirable. It is worth noting, however, that the definition of *Adverse Outcome* does not necessarily imply that these outcomes are "bad", instead it only defines what the trustee views as undesirable.

In addition to the outcome, some by-products (referred to as *Symptoms* by Riegelsberger et al. [139]), such as event logs or traces, will be generated during the execution of the actions. These are captured as a subclass of *Signal* which is used to denote the idea that the *Trustee* has to send evidence to the *Trustor* to enable them to make trust decisions.

We have specified trust relationships in terms of the expectations from the actions of the trustee. However, this does not specify whether or not the trustee can perform those actions. This can be captured using the concepts *Ability* and *Motivation* [139]. Riegelsberger et al. [139] specify that a trustee must have the ability and motivation to perform the expected actions otherwise no trust can be established. *Ability* captures the capacity and competence of the trustee to perform certain actions while *Motivation* captures the incentives in place that encourage fulfilment of the actions. However, it is argued that motivation can not be applied to non-cognitive entities [123, 139] and, therefore, it is ignored in this model.

In summary, the conceptual model for system trust, illustrated in Figure 2.5, is as follows. A *Trustor trusts* a *Trustee* who *has Ability toPerform* an *Action*. Such an *Action* is *performedOn* an *Asset* (e.g. *Resource* or *Data*) which is owned or controlled by and is *valuableTo* the *Trustor*. Upon performing an *Action* a number of *Outcomes* are produced together with remnants called *Symptoms*, which can be used to signify the presence of *Trustworthy Actions* — subsets of actions which the trustee considers to be indicative of expected behaviour.

2.3 Trusted Computing Provides Technical Mechanisms for Trustworthy Operation

In Section 2.1.2 we discussed a conceptual model of a system architecture and in the previous section we discussed how the notion of trust can be captured in the context of a system. In this section we investigate how the two concepts are related by reviewing the technical mechanisms, provided by trusted computing, for enabling trustworthy operation.

Every year billions of dollars, in equipment and human resource, are spent on preventative measures against malware attacks and repairing the damage caused by them[1]. A response to this threat focuses on using technologies such as virus and rootkit detectors, intrusion detection systems and firewalls that use signature-based or file heuristics to detect malware and prevent it from infecting computers. Despite these efforts, computer systems still continue to be compromised. This suggests the existence of a race-condition between the "good", owners of the computers, and the "bad", producers of malware, in which the "bad" are always a step ahead. To

[1] 2007 Malware Report: The Economic Impact of Viruses, Spyware, Adware, Botnets, and Other Malicious Code, Available at http://www.computereconomics.com/page.cfm?name=Malware 20Report

overcome this, it is required that a system operates in a trustworthy manner. This is the aim of trustworthy systems. But what exactly are these systems and how do they achieve this?

2.3.1 So What are Trustworthy Systems?

The RFC 4949 [156] defines a trustworthy system as one "that not only is trusted, but also warrants that trust because the systems behaviour can be validated in some convincing way, such as through formal analysis or code review". Combining this with the earlier definition given in Section 2.2.1, we define a trustworthy system as:

> ***as a system that always behaves according to the expected manner for the intended purpose and allows relying entities to verify its behaviour against their expectations.***

Based on a characterisation of a trusted entity provided by Proudler [135], we specify that trustworthy systems have the following characteristics:

1. a trustworthy system can be correctly and unambiguously identified,
2. a trustworthy system operates unhindered,
3. the trustor will either have experience or trust someone who has of consistent previous good behaviour in a trustworthy system, and
4. a trustworthy system provides mechanisms to demonstrate its ability to protect itself or the data stored on it.

These characteristics are achieved by including a number of components in a system architecture and processes that ensure that these components work together to enable predictable and verifiable behaviour. This is provided by trusted computing mechanisms.

2.3.2 Trusted Computing as Foundation for Trustworthy Systems

Trusted computing is a different approach towards overcoming security problems. It is based on a realisation that the majority of attacks happen in software and that "software alone can not protect itself" [25] and, therefore, provides cryptographic mechanisms that help to securely identify the configurations of hardware and software

on a system. The configurations can be securely reported to any other entity wishing to rely on the system, allowing such entities to determine whether to trust the system or not.

On a computing platform comprised of multiple hardware and software components how can one be assured of the integrity of the data on which a particular component operates? If trusted computing helps to provide accurate measurements of the software configuration of a platform, how can we be sure that the measurements presented are of high integrity? How can we even trust that the measurements are coming from a genuine component and that the component is not under the influence of some malicious entity? Furthermore, trusted computing makes use of cryptography, which relies on the secrecy of keys used in encryption/decryption and digital signatures. How does trusted computing ensure that the keys remain secret?

To answer these questions, we describe the building blocks of trustworthy operation provided by trusted computing and investigate how these help to improve platform security. We further describe how these capabilities have been combined with mechanisms such as virtualisation and PKI to enable trustworthy system architectures [94, 31] and investigate challenges with respect to reasoning about trust-related properties. The results of the investigations carried out here, together with the conceptual model discussed in Section 2.1.2, are used as the foundations of the semantic model presented in Chapter 4.

2.3.3 The Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) is a functional specification for a device that provides cryptographic functions to a computer system. The TPM specifications [165] specify that the TPM should be cheap to manufacture, securely installed on a platform in such a way that it cannot be influenced by software running on the platform on which it is installed, and be tamper resistant against minor hardware attacks.

The internal structure of a TPM comprises of various components including a random number generator, an execution engine, volatile and non-volatile memory, and an array of 20-bit storage areas called platform configuration registers (PCRs). Data is stored in the PCR through the *extend* operation, described in Section 2.3.5, so that each operation on a given PCR results in a 20-bit hash value of the contents and data to be stored.

Continued integration of virtualisation and trusted computing technologies means that the TPM would have to be shared among multiple domains running on the same

host. To enable this, virtual TPMs (vTPMs)[15, 147, 42] have been proposed as resources on a virtualised system which mimic TPM interfaces, thus allowing virtualised systems to make use of hardware-based protection mechanisms. However, it is still not clear how their use impact security and trust establishment in such systems. For example, vTPMs are expected to provide similar functionality to a TPM without impeding key processes, such as migration, that make virtualised systems desirable [33]. However, a vTPM is faced with challenges, such as how to protect secrets from software running on the same platform, which a hardware-based TPM was designed to overcome. This is a general problem with most trusted computing concepts, which can be attributed to a lack of a systematic way for understanding the extent to which these technical mechanisms influence trust in a system. We address this issue in Chapter 4 by proposing a semantic model that captures key concepts and their properties.

2.3.4 Roots of Trust as Foundations of Trust in a System

Trusted computing enables one entity to make an informed decision about the trustworthiness of another entity. But what makes one entity believe what another entity says? What prevents the challenged entity from lying about its trustworthiness? For example, Challener et al. [25] note that it is hard for software to prove that it can be trusted because there is nothing trusted to provide that answer. So how does trusted computing overcome this?

In the context of trusted computing, trust relies on some kind of reference. The trusting entity can either refer to the good experience they have had with the entity being trusted or they trust some other entity that can vouch for the trustworthiness of the entity being challenged. This is enabled by the notion of *root of trust*, which is implicitly trusted for some aspect of the system such as measurement of integrity or storage of secrets. The following roots of trust are provided in trusted computing:

1. The root of trust for measurement (RTM) provides a means for accurate generation of integrity measurement of the software running on the platform. The RTM can be classified as either static or dynamic. Static RTM provides an immutable root that is established once and can only be reset when the entire platform is reset. Dynamic RTM on the other hand can be established at any point, without resetting the entire platform, and as many times as necessary.

2. The root of trust for storage (RTS) provides accurate storage of integrity measurements and enables integrity and confidentiality protection of data through cryptographic mechanisms such as sealing.

3. The root of trust for reporting (RTR) provides capabilities that enable accurate reporting of integrity measurements. This is done through the use of an endorsement key and endorsement credential which are both provided by the manufacturer to demonstrate that the TPM meets the specifications and that it is bound to a specific platform.

The use of these roots results in the creation of a *chain of trust*, in which a particular root of trust is at one end of the chain. However, the concept of trust is limited to a single relationship between entities in the chain. For example the chain of trust obtained during the boot process is based on every component measuring the next component in the chain before passing control to it [59]. Presti [133] and Schmidt et al. [153] acknowledge this problem and propose a chain based on a tree structure and propose extensions to the TPM command set to enable the generation of such data. However, a tree structure limits the types of relationships that can be captured between entities in the chain. For example, there is no straightforward way of relating the leaves of a particular branch (other than the relationships to their parent). In Section 6.4 we demonstrate how a model based on a graph structure addresses this problem.

2.3.5 TPM is Secure Against Software Attacks

As mentioned earlier, a majority of attacks happen in software and the TPM is designed to be resistant against software attacks. But what exactly makes this possible?

First, though the TPM is a functional specification that can be implemented in software or hardware, a hardware-based TPM "gives useful guarantees of trustworthiness" [164]. The hardware chip has an internal execution engine that allows it to execute its functionality inside the chip and only exposes the results once security sensitive data has been secured, for example after encryption. This enables it to operate without interferences from software running on the platform.

Second, the TPM provides an immutable root of trust. This means that not even software running on the same platform can compromise it. This root of trust can be extended by measuring the next component in the boot chain and securely storing the measurement before passing control to it. Using this technique a system can be booted up in two ways: authenticated boot or trusted boot. Authenticated

boot ensures that each component of the boot process is integrity measured and the measurement is securely recorded but, unlike trusted boot, does not enforce that the components be of high integrity. On the other hand, trusted boot ensures that the system only starts up when all the components are of high integrity. This is normally enforced by comparing the integrity values to a list of known-good configurations or a whitelist [72].

Finally, the TPM is designed to prevent directly writing to it. This prevents software running on the platform from directly changing the values that are stored in the TPM. Instead the TPM allows changes to the PCRs to be made only through the *extend* operation, which appends data to the existing contents of a given PCR index using a cryptographic hash operation. The *extend* operation relies on the non-commutative nature of cryptographic hash to ensure that only the same data, extended in the same order, can produce a particular state of the PCR. What all this means is that rogue software cannot directly insert malicious values in the PCRs.

2.3.6 Secure Storage: Secrets Never Leave the TPM

One of the main design goals of trusted computing is the protection of security-sensitive information such as keys from theft or use by malicious entities. This is provided by a secure storage functionality in combination with public key authentication. That is, the random numbers are generated inside the TPM together with public keys which are stored in the TPM and always encrypted, with another key stored inside the TPM, when being transferred outside the TPM [25]. This is made possible by the key hierarchy available to the TPM, where one key called the storage root key is generated once and never leaves the TPM but allows an arbitrary number of keys to be generated and stored outside the TPM such that they are ultimately protected by the storage root key.

Though the storage root key is stored on the same platform as the encrypted data, it is still secure because trusted computing is designed to ensure that the SRK can never leave the TPM. Its use relies on knowing some authorisation data. This means that to get access to the secret keys stored under the SRK, one has to provide authorisation data which is only known by the owner of a platform. In addition, all the keys in a given hierarchy are locked to the SRK so that changing the SRK, e.g. by taking ownership, results in the loss of the storage root keys — making all keys linked to it inaccessible. This functionality can be observed in Microsoft's BitLocker[2] application which encrypts the disk and stores the key in the TPM.

[2]http://www.microsoft.com/windows/windows-vista/features/bitlocker.aspx

2.3.7 Components are Integrity Measured Before Execution

An integrity measurement of a platform component is defined as the cryptographic digest or hash of the component [129]. This measurement may include the component's executable files, configuration files or pieces of data, such as parameters or environment variables, that may affect the behaviour of the component. The measurement is obtained through an integrity measurement process which is defined by the TCG as a "process of obtaining metrics of platform characteristics that affect the integrity (trustworthiness) of a platform." This process has been implemented in IMA [148], which is now part of the Linux Kernel, and PRIMA [82], which enables integrity measurement using information flow control provided by SELinux [106].

If this process occurs on the same platform hosting other potentially malicious software and hardware components, what integrity assurance does trusted computing provide? What prevents a malicious entity from forging these measurements? The first part of the answer lies in the property of cryptographic hash — that of being able to produce a unique digest for every one bit difference in data, assuming there are no collisions.

The second part lies in the use of roots of trust — particularly the root of trust for measurement. The RTM is trusted to record the integrity measurement of the next component in the boot process before passing control to it thus ensuring that each component is measured before being executed [59]. This is a key property which implies that components cannot lie about the state they are in.

2.3.8 Attestation Enables Trustworthiness Verification

One of the main aims of trusted computing is to enable one entity to make informed decisions about the trustworthiness of another entity. This is enabled through a reporting mechanism called *attestation*.

Attestation is a means of locally or remotely authenticating a platform's software configuration. Cooper and Martin [31] describe it as "a powerful security primitive that allows a party to obtain assurance in the correct operation of a system that is beyond their physical control." This process involves the generation of digitally signed integrity measurements by the challenged platform and a subsequent verification of both the signature and the trustworthiness of the measurements contained in the package.

Attestation based on hash digest and of large components such as operating systems and virtual machines has, however, been noted as being hardly meaningful as

these components have millions of lines of code and change regularly, making it hard to keep a record of all the possible digest values [40, 31, 43, 146, 64]. The fundamental problem is that such an approach helps to determine the existence of a particular binary on the system rather than its behaviour [5, 64]. A number of proposals have been made to address this problem. England [43] proposed the use of specialised read-only operating system images as a means of overcoming this problem, while McCune et al. [108, 107] insist on using mechanisms that reduce the TCB to improve attestability of software. Of particular interest to this dissertation are those mechanisms that concentrate on high-level properties of a system including property-based attestation [146], behavioural attestation [93, 5] and semantic remote attestation [64].

Sadeghi et al. [146] propose property-based attestation as a way of mapping low level configurations to system properties and thus avoiding disclosure of configurations to the challenger. This is achieved by employing a third party, which is trusted to perform the translation between low-level configurations and system properties. The configurations considered in property-based attestation could be considered as part of evidence in our models, presented in Chapters 4 — 6, and, therefore, our proposal could be integrated into the processes performed by the third party. Furthermore, our proposal allows these properties to be verified enabling both the third party and those relying on the properties to gain confidence in the mapping.

Haldar et al. [64] propose semantic remote attestation, in which a trusted virtual machine is used to monitor the security policy attached to software and to perform semantic analysis to verify the behaviour of the software. However, semantic remote attestation still relies on binary attestation as the virtual machine and other software it relies on must still be verified using binary attestation. Li et al. [93] propose behavioural attestation, which instead focuses on the behaviour of the security policy. Alam et al. [5] build on this approach and propose model-based behavioural attestation, where analysis is performed on a policy model rather than the software or hardware of a system.

2.3.9 TCG Integrity Schemas: A First Step Towards Evidence Collation

The TCG Infrastructure working group recognises that for one to establish trust in a platform, it is important to know the origins, conditions and history of components used to construct the platform. They define an architecture [163] and a number of specifications aimed at providing mechanisms for collecting and validating the integrity of platforms in an interoperable manner.

They define an integrity management model consisting of creation, collection, communication, collation and evaluation phases. They further distinguish runtime from reference measurements, such that runtime measurements occur during the creation and collection phases and are evaluated against reference measurements that are prepared following the four phases. As illustrated in Figure 2.6, during the creation phase, a component, referred to as a *creator*, measures components on the platform and generates integrity measurements (IMs). The integrity measurements are collected and formatted into integrity reports (IRs) by a component referred to as a *collector*. The resulting integrity report is then sent to the verifier which uses them in trustworthiness evaluation.

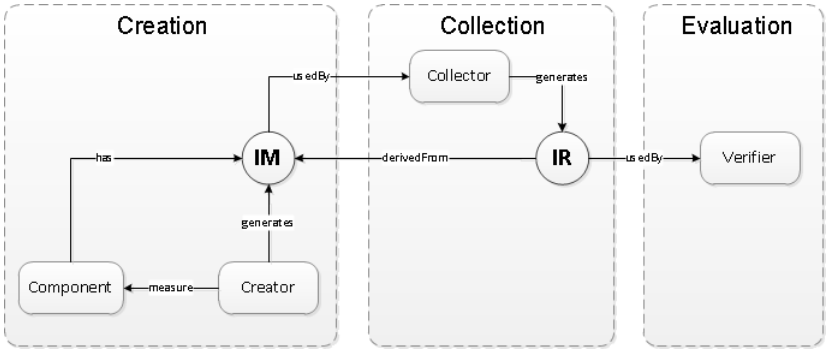


Figure 2.6: TCG Integrity Verification Mechanism

Since the evaluation phase may occur in a different domain from the creation and collection phases, it is important that the data sent over is correctly interpretable to the verifier. To support this, the TCG provides a number of schemas that define the format of the data exchanged between the target platform and the verifier. The schemas are useful for capturing the integrity measurement of the components on the platform. However, they cannot be used for capturing a more comprehensive description of a platform state. We believe that integrity measurement is essential for trustworthiness evaluation, but is not sufficient. This is because evaluation is restricted to checking the presence of certain components. What is needed, instead, is a way of not only determining the components present, but also the relations among them, the settings of the environment in which they exist, and the data they use or produce.

2.4 Reasoning about Properties of Systems

In the previous sections we discussed the challenges in defining trust in complex systems and how an architecture helps to manage this complexity. In this section we investigate attempts made towards addressing this issue. We begin with the more general issue of using an architecture to determine the properties of a system before discussing efforts invested towards reasoning about properties of trustworthy systems.

2.4.1 Runtime Architecture Discovery and Verification

Architectural design ensures that errors are identified early. However, even when an architecture is error-free its implementation may introduce errors as well. To ensure that the implementation follows from the architecture requires monitoring a running system to discover its runtime architecture and validate it against its architectural design. This has been demonstrated in ArchJava [7], DISCOTECT [181] and in [69].

Aldrich et al. [7] propose ArchJava as an extension to Java that unifies software architecture with implementation and guarantees communication integrity between an architecture and its implementation. The ArchJava approach integrates architectural constraints within the code. This approach, also taken by Dong et al. [38], has the disadvantage that the architecture within an ArchJava program cannot be reused in another program without rewriting it. In contrast, DISCOTECT [181, 151, 152] separates the architecture by using an architectural style mapping file, which is then compiled and used by a runtime engine, which monitors events from a running system and converts the events into an architecture following the architectural style specified. Huang et al. [69] propose recovering the software architecture from component-based systems at runtime in order to change the system at runtime. These mechanisms are useful for capturing architectures at the program level, rather than at the system level. The focus is also limited to functional aspects of the program without considering how non-functional characteristics such as security can be addressed.

2.4.2 Reasoning with Security Logic

The idea of using logic in the analysis of security goes as far back as the early 90s. The 1990 paper on a *Logic of Authentication* by Burrows, Abadi and Needham [23] led to what is commonly known as BAN logic, which enables the specification and reasoning about beliefs of trustworthy parties involved in an authentication protocol. Several extensions and variations to BAN logic have been proposed including GYN logic [57] — a logic of authentication that enables key secrecy across domains,

establishment of beliefs in key secrecy without jurisdiction [86], and an extension focusing on certificate processing within PKI settings [162]. However most of this work was restricted to specification and analysis of security protocols. A slightly different approach can, however, be observed in the work on *security logic* [56] proposed as a means of specifying and reasoning about security policies. The logic is based on modal logics, enabling specification of beliefs and knowledge about policy enforcements in a system.

Recent efforts [35] have seen the ideas being applied to reasoning about security of an entire system, not just protocols. Using insights from the protocol composition logic (PCL) [32, 34], Datta et al. [35] propose logic for secure systems (LS2) for modelling machines and reasoning about their security state and further demonstrate its applicability by characterising some trusted computing primitives. LS2 is designed with a syntax that allows modelling of machine primitives such as shared memory, cryptographic operations and network access and semantics based on a proof system that enables reasoning about security properties of a system.

Demolombe [36] proposes a framework of modal logic to reason about trust. The framework aims to define facts that support trust (observations and reputation), define rules for deriving consequences and enable the use of this information in the trust decision process. The author defines trust in terms of epistemic properties such as sincerity, completeness, validity and deontic properties such as obedience, honesty and ability, and develops axioms that allow the deduction of the trustworthiness of an agent. Similar attempts were made by Chen et al. [26] who defined formal semantics using predicates such as "integrity", "trusted" and "measurement", and inference rules to reason about the behaviour of an entity. Jøsang and Knapskog [85] propose a model consisting of an evidence space, opinion model and subjective logic to quantity and reason about trust in IT systems.

In order to determine whether a utility computing infrastructure will meet the required security requirements and therefore trust in its behaviour and ability to protect data, potential customers would need to verify that the infrastructure has specific (vulnerability-free) security mechanisms that will achieve the desired goals. One way of achieving this is to enable reasoning about security state of the infrastructure at different levels of abstraction. Monahan [114] demonstrates how a logic-based modelling approach can be used to enable both providers and customers to obtain insights about the security state of a utility infrastructure. He further highlights the importance of abstraction, stressing that it is intractable to derive and extract useful information from complex systems without abstraction.

2.4.3 Reasoning about Properties of Trustworthy Systems

The need to reason about properties of trustworthy systems has been identified by various authors. In his doctoral thesis, Rohrmair [142] proposed the use of CSP to model and verify parts of a trustworthy system architecture. More specifically, he models protocols for authorisation, chain of trust generation and session caching mechanisms. The focus of his work is limited to protocols, which we believe does not reflect the overall properties of trustworthy system architectures. Similarly, Lyle [98] applied CSP semantics towards verifying integrity measurement logs. He argues that attestation reports the execution integrity of a system rather than behaviour of the system and proposes an approach for reporting platform behaviour by combining system behaviour models with integrity measurement logs. However, his work is limited to integrity measurements and only describes PCR usage. The CSP models he proposes are based on extend operations and how these can be used to establish whether an attesting system used PCRs in some expected manner.

Our approach covers much broader aspects than the work of Rohrmair and Lyle. First, we take an architectural approach which allows us to separate different aspects of a trustworthy system. We then build models which are designed to capture each of these aspects. The models and protocols proposed by both Rohrmair and Lyle can be considered as subsets of our models. More specifically, the CSP model proposed by Lyle and part of the protocols modelled by Rohrmair are included as part of the *Trust Services*, discussed in Section 5.2.3. Furthermore, the focus of our work is on architectures, which differs from Rohrmair, who focuses on protocols, and Lyle, who focuses on attestability of web services.

Similar work on reasoning about trust has also been investigated by Fuchs *et al.* [51] who formalise the notion of trust. The authors define a formal notion of trust as related to ICT systems and propose an approach for integrating it into a security modelling framework. Their approach serves as a good way of expressing trust properties and assumptions that affect the security of a given system. However, the authors do not consider the kinds of evidence necessary to verify these properties nor how constituents of a system bring about these properties. Furthermore, though the notion is intended to support multiple perspectives, it is not clear how the formalism distinguishes these perspectives, other than through the types of properties specified. Our work makes use of an architectural approach to enable multiple perspectives and a building block approach to enable capturing how properties are induced by composing architectural elements.

2.5 Summary

The notion of trust can be used to mean different things in different contexts. Its use in the context of a system makes it particularly confusing both in terms of what it means to trust a system and to establish whether or not to trust a system. To overcome this challenge, in this chapter, we have provided a definition for trust that will be used throughout the dissertation and developed a conceptual model that captures this notion in the context of a system. We have also discussed how trustworthy system architectures make use of technical mechanisms to enable trustworthy operation. However, establishing how these technical mechanisms influence trust, the nature of evidence necessary to determine this, and how this evidence can be used is a significant challenge. We have investigated approaches towards addressing this problem, and related problems, and discussed their shortcomings. In the following chapters, we use the models and lessons learnt in this chapter to build solutions to some of the problems identified.

Chapter 3

Architecture-based Framework for Trust-properties

In Chapter 1 we discussed how computer systems, of various kinds, are entrusted with more and more sensitive information which if modified, leaked or corrupted may have undesirable consequences. As a result, it is essential to establish how *trustworthy* a system is before it can be entrusted with sensitive data [55, 77, 60]. However, trust establishment, especially in remote systems, still poses a significant challenge and a hindrance towards wider adoption of distributed systems such as those based on cloud computing [125, 87].

This challenge can be attributed to a number of factors. First, as discussed in Section 2.2, the notion of trust can be used to refer to different things, especially when used in different contexts. And indeed, even after decades of research and debate, there is still no consensus on its definition [67, 180, 90] and others [123, 139] argue that such a notion cannot be applied to non-cognitive entities. Second, continued demand for increased speed, features, ubiquity and accessibility of computer systems has led to an increase in their complexity. And finally, even though technical mechanisms such as trusted computing can be used to enable trustworthy operation, it is not clear how their use influences trust in a system. Consequently, the properties or assumptions upon which trust is based are not made explicit.

An integrative approach that enables trust to be defined in the context of a system, provides mechanisms for managing the complexity and a way of identifying properties and assumptions upon which trust is based is essential for addressing these issues. Towards this end, we propose a framework that uses an architectural approach to provide abstractions that serve as a means of managing complexity and providing views through which reasoning about trust can be achieved.

In the next section, we describe three scenarios that illustrate the need to establish trust in a system. These scenarios are used throughout the dissertation both for analysing problems and evaluating solutions. In Section 3.3 we define trust-properties and develop a mechanism for classifying them. This is followed by the identification of architectural aspects. These aspects are based on the architecture model discussed in Section 2.1.2. In Section 3.6 we identify the possible mapping between the classes of properties and architectural aspects that could be used to reason about properties in each identified class. In Section 3.7, we describe how this framework motivates the development of models that capture specific aspects of a system, and which are developed in later chapters of this dissertation. We summarise the findings of this chapter in Section 3.8.

3.1 Scenarios

In this section we describe three scenarios, including a password manager, a trusted grid platform and a collaborative system. The first scenario was elicited from a use case on secure web authentication [22] while the second and third scenarios are based on operational systems that are offered by the Oxford e-Research Centre (OeRC).

3.1.1 Scenario I: Password Manager

Username/password mechanisms are one of the most common means of authentication. Most online services such as electronic banking, social networking and email require users to authenticate using username/password combinations. Each service also places certain constraints on the format or character combinations used for passwords. As a result, users cannot remember such passwords and thus require mechanisms for managing their passwords.

We consider a scenario of a password management system [22], illustrated in Figure 3.1, in which a user relies on a password manager (passman) to generate and store passwords. *passman* also enables users to either copy the passwords to the particular service or to supply the password directly to the authentication service. A user interacts with the *display* component which relays requests to *passman* via a display driver. *passman* is comprised of a password generator, a storage system and a password provider. The password generator generates new passwords which are stored in the storage system. Upon the user requesting a password, the password provider obtains the passwords from the storage system and passes it to the user through the display or sends the password directly to the authentication service. Meanwhile,

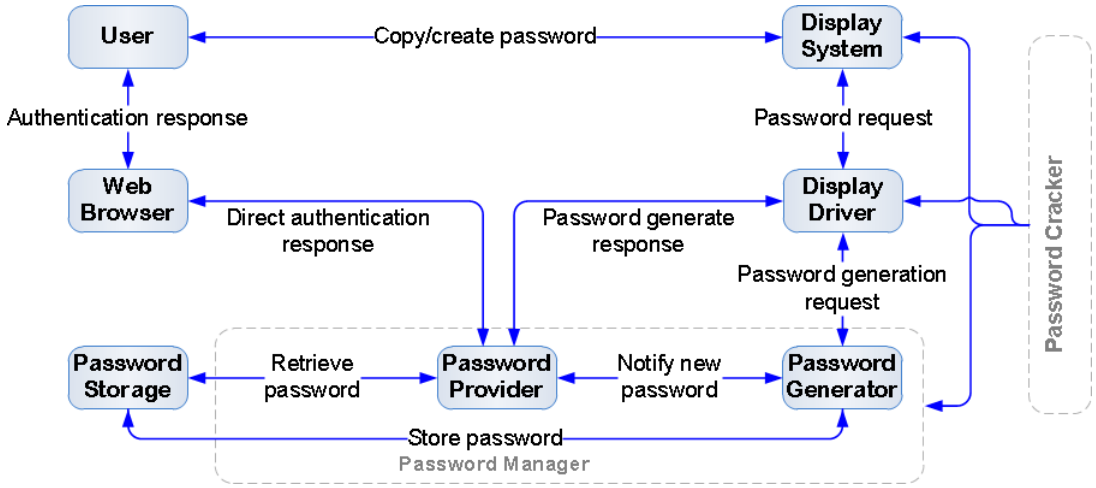


Figure 3.1: An architecture of a password manager

another application, *passcracker*, runs on the same system and attempts every way possible to deduce the passwords used by the user.

3.1.2 Scenario II: A Trustworthy Grid Platform

Grid computing enables multiple heterogeneous resources distributed across multiple organisations and administrative domains to be connected into a set of resources capable of scaling to Internet-size environments [88]. The design principles for grid computing enable commodity-like provision of compute and data storage, a paradigm which has been better adopted by various cloud computing models [49]. However, grid computing is faced with several trust and security challenges [31, 105, 74]. These can be attributed to a number of factors, including: a lack of mechanisms for ensuring consistent policy enforcement; a lack of mechanisms for securely managing configurations [72]; and a lack of mechanisms for providing proofs about their behaviour [59].

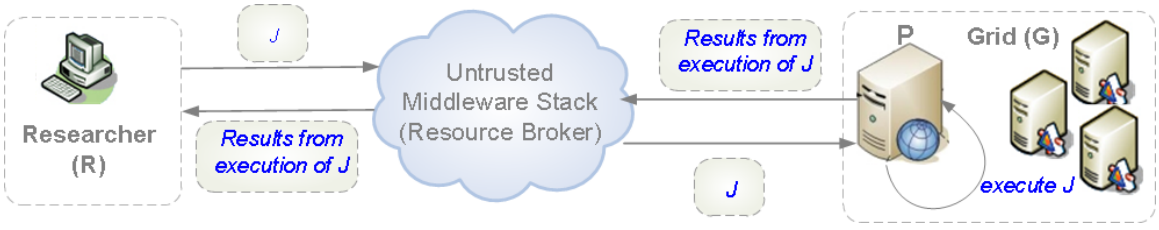


Figure 3.2: A researcher using a platform that is part of a grid infrastructure

We consider a scenario, illustrated in Figure 3.2, in which researchers registered with the National Grid Service (NGS) [122] make use of resources provided by a grid

infrastructure, G. G is a composition of resources P from multiple administrative domains including colleges and departments within the university and other centres that are part of NGS.

A researcher, R, has a job, J, that runs simulations to determine the influence of a new drug on a certain population of individuals. Upon requesting resources to run J, she is assigned an available resource P to execute J. Due to the sensitivity of the job and the data that it handles, R requires that the data used or produced must maintain integrity and be kept confidential. In addition, R needs to ensure that the algorithms used as part of the simulations are the same ones that she deployed in J. Therefore, R needs to establish that the system has sufficient capabilities to ensure that these constraints can be upheld and requires evidence that these capabilities are in use.

3.1.3 Scenario III: Trustworthy Collaborative System

With the growing influence of e-Research, a substantial quantity of research is being facilitated, recorded, and reported by means of cloud computing and various e-Infrastructures. Systems such as UnaCloud [127] and SciCloud [161] provide cloud infrastructure for research. Developments in Open Source software such as Eucalyptus [124] and OpenStack [134] have also contributed to the adoption of cloud computing in e-Research.

As part of continued improvement in the service offerings, the Oxford e-Research Centre (OeRC) has introduced a Eucalyptus-based cloud infrastructure for use by researchers. We consider a scenario in which multiple virtual organisations (VOs) engage in collaborative research, such that researchers from one virtual organisation may need to share data and computational and storage resources in order to facilitate their experiments.

The collaboration works as illustrated in Figure 3.3. A researcher, $R1$, from virtual organisation 1, $VO1$, prepares an experiment (step 1) by installing and configuring the operating system, applications, their dependent libraries, input data and algorithms. Security configurations, such as SELinux policies, are also configured during this stage. To ease deployment, the experiments are deployed as virtual machines on the cloud infrastructure provided.

As a contribution to the project, $R1$ may publish this virtual machine — an initial experiment setup — to the central image repository (step 2). Alternatively, $R1$ may decide to deploy the virtual machine onto the infrastructure to run some experiments (step 3.A). When the experiments have completed, $R1$ publishes the virtual machine

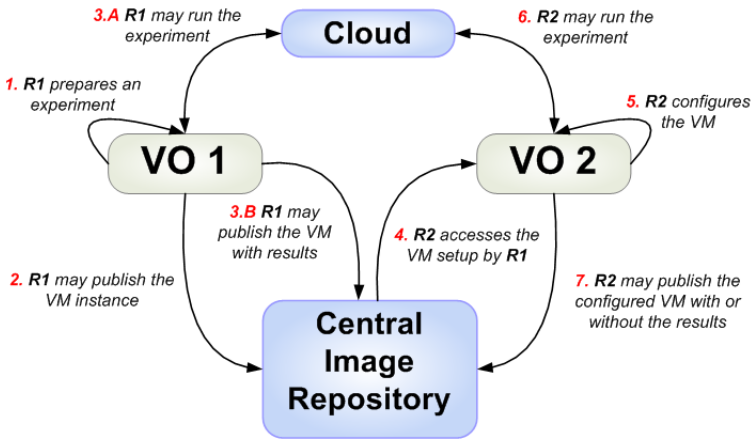


Figure 3.3: Workflow showing how a virtual appliance is shared in a cloud

together with results of the experiments (step 3.B) as a virtual appliance — a pre-configured virtual machine image ready to run [121].

$R2$ from $VO2$ may design an experiment based on either the initial experiment setup published by $R1$ (step 4) or the results generated from running $R1$'s experiment (steps 5 and 6). Since $R2$ is from a different virtual organisation, the downloaded virtual appliance needs to be configured to conform to the policies of $VO2$. At this stage, $R2$ may choose to publish this modified virtual appliance (specifying how it has changed) or choose to deploy it to the infrastructure in order to run the experiment and publish it after the experiment is completed running (step 7). Before setting up the experiment, $R2$ needs to be confident that the virtual appliance (downloaded from the central repository) when deployed will preserve the integrity and confidentiality of the experiments.

3.1.4 Analysing the Scenarios

The scenarios described above point to the need for some entity (system or human) to establish trust in another system. They illustrate a number of issues regarding trust establishment, some of which are discussed below.

1. *What does it mean to establish trust?*

   The scenarios described above have various entities with distinct types of relationships among them. In the grid scenario, a researcher will have a computer which she uses to interact with the grid. In the password manager scenario, the user interacts with the password manager application, *passman*, to request new passwords. In the collaborative system, researchers interact with platforms

that create virtual machines as well as virtual machine instances running on the system. In these scenarios, coming up with a consistent meaning of trust is challenging. This is because of the level of complexity involved in each system. A grid system, for example, comprises of multiple platforms each of which may be involved in providing a service to the researcher.

2. *From whose perspective is this trust?*

   The systems presented in the above scenarios are depended on by multiple entities. System administrators rely on these systems to enforce policies they set and monitor the usage of resources while users (e.g. researchers) rely on the systems to provide the necessary services under appropriate conditions. In the third scenario, for example, project managers who define the requirements for each experiment expect that the system will satisfy these requirements while software developers who create the software used (e.g. execution environment) for the experiments expect the runtime environment to behave in a certain way. This means that each entity will have some *expectations* from the system with respect to the constraints they wish to place on it or its environment. In establishing trust, it is crucial that the entities relying on the system as well as their expectations are made explicit.

3. *How can trust be established amid complexity?*

   Each system used to support the scenario has to be designed with some quality attributes such as reliability, performance and availability in mind. These attributes are achieved by including a number of components that are connected together using complex communication infrastructure. For example, to provide high availability, multiple platforms within the system can be set-up so that when one platform fails, another will be ready to take over the tasks that were assigned to the failed platform. The commodity-like nature of a grid computing-based system implies that the system has to be designed with capabilities to manage the dynamism of constituent platforms resulting in yet more complexity. Therefore, any entity wishing to establish trust in such systems has to do so in the midst of complexity.

4. *How does the use of abstraction layers influence trust establishment?*

   The complexity described above is generally dealt with using abstraction mechanisms, where several abstraction layers are defined in a hierarchical manner such that the highest layer is less detailed than the layer below it. For example,

the three cloud computing layers, Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Application as a Service (AaaS) [112] are a means of abstracting away from the complexity in the operation of the underlying infrastructure. Similarly, the grid scenario described in Section 3.1.2 provides middleware which hides all the complexities in job scheduling, submission and migration, and credential management. The use of these mechanisms, however, introduce the challenge of how to establish trust while working at these abstraction layers.

The scenarios discussed above involve complex systems that are depended upon by multiple stakeholders. These systems make use of abstraction mechanisms to simplify interactions. This implies that stakeholders have to define and establish trust in such systems in the presence of these abstractions. While the challenge of establishing trust in a system has received considerable attention, not much has been directed towards understanding how this can be achieved in the presence of these abstraction mechanisms and the implications of doing so.

In the rest of this chapter, we present a framework that enables us to frame the problem of trust establishment in terms of expectations of the relying parties. The framework defines these expectations as *trust-properties*, classifies them and proposes a mapping between the property classes and architectural aspects that are best suited for reasoning about each class of properties.

3.2 Framework Overview

In this framework, given an entity, X, and a system, S, X's trust in S depends on the behaviour of S, the purpose for which X intends to use S and S's ability to satisfy properties deemed essential by X under a given set of assumptions. To establish trust, X begins with a set of assumptions. For example, X could decide to assume that *systems manufactured by some manufacturer, M, always behave according to her expectation.* Given a system, if X can determine that the system is manufactured by M, then X can rely on that system. However, to increase the confidence that X has in S, X may collect evidence about S's behaviour and use this evidence to establish whether or not properties of interest are satisfied. These properties are linked to assumptions, so that satisfying a given property could discharge (i.e. eliminate the need of) one or more assumptions. S will, therefore, be considered trustworthy with respect to an intended purpose if evidence collected about S's behaviour supports properties defined by X under reasonable assumptions. By so doing, the framework

helps to make explicit the properties and assumptions against which trust in a system is based.

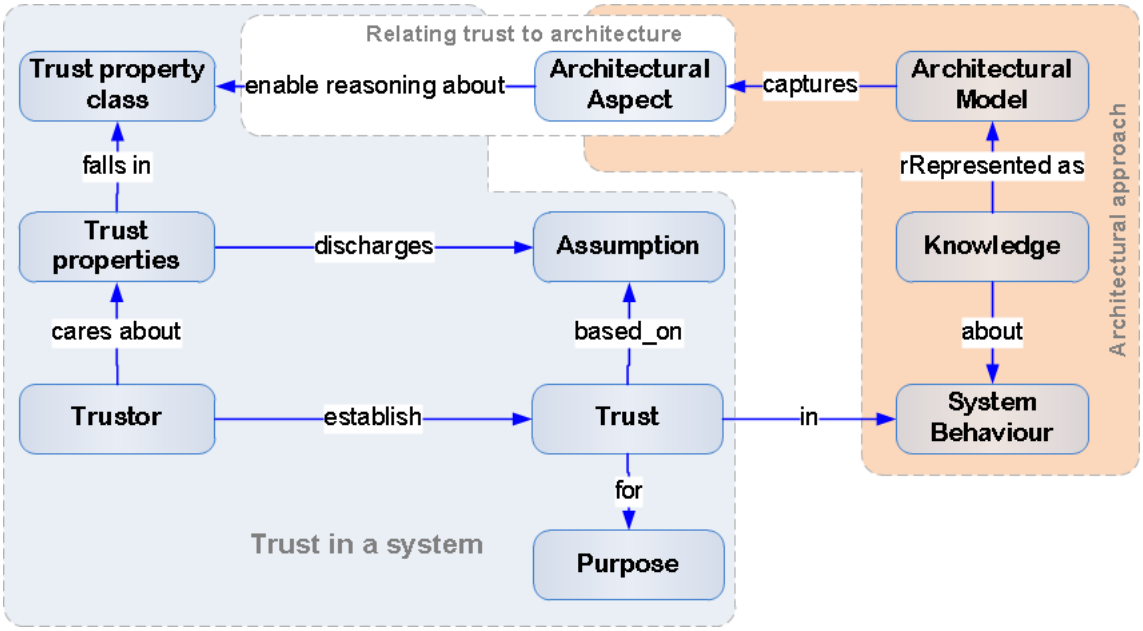


Figure 3.4: Structure of the architecture-based framework

3.2.1 Objectives

The framework is designed with the following objectives in mind.

1. Provide means of defining trustworthiness with respect to a system architecture.
2. Identify and classify the properties that signify trustworthy behaviour.
3. Identify the aspects of an architecture that help to reason about the identified classes of properties.
4. Guide the structure of mechanisms for capturing each architectural aspect.
5. Guide the format of specifications for trust-properties that can be reasoned about using each aspect.

3.2.2 Approach

We note that an architectural approach is useful for managing the complexity of computer systems [183]. It provides abstraction mechanisms that enable non-essential details about a system to be hidden, while leaving only those that are essential for

reasoning about a particular set of concerns. However, the choice of these abstractions and their relationship to trust establishment is something that has not been explored. The framework addresses this issue by using the trust characterisation discussed in Section 2.2 and the architectural model discussed in Section 2.1.2 to identify classes of trust-properties and architectural aspects. The two are then mapped so as to identify which aspect is best suited for addressing a given class of properties.

We realise that trust is a cross cutting concern, which may require looking at a number of aspects (or indeed all aspects) in order to reach a decision about whether to trust or not. The framework's support for this is based on the idea that any given trust decision can be broken down into establishing whether a system can guarantee one or more properties under certain assumptions. Each property could then be determined by looking at some aspect of the system.

In the next section, we define trust-properties, identify four main classes of trust properties and develop a mechanism for classifying trust properties into one of the identified classes.

3.3 Trust Property Definition

In Section 2.2.2 we presented a model in which trust in a system is defined in terms of the actions that the system can perform. This model enables the expectations of stakeholders to be expressed in terms of actions that a system can perform or results that can be obtained when the system performs these actions. These expectations can be considered as a subset of stakeholder concerns — defined as "those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders" [39]. However, based on the TCG definition of trust, i.e. being able to verify the behaviour, the expectations must be verifiable. This dissertation makes use of the term trust-property to capture concerns that can be checked against the system in question.

We define a trust property as:

> ***an assertion about actions and/or results expected from a system or environment in which it runs, that contribute to a consistent behaviour of the system.***

Trust properties are expressed from the point of view of a stakeholder, and thus can be distinguished from trust-warranting properties defined by Bacharach and Gambetta [12], which are concerned with indications that could lead one to trust a system. In other words trust properties are expressed by stakeholders and are checkable

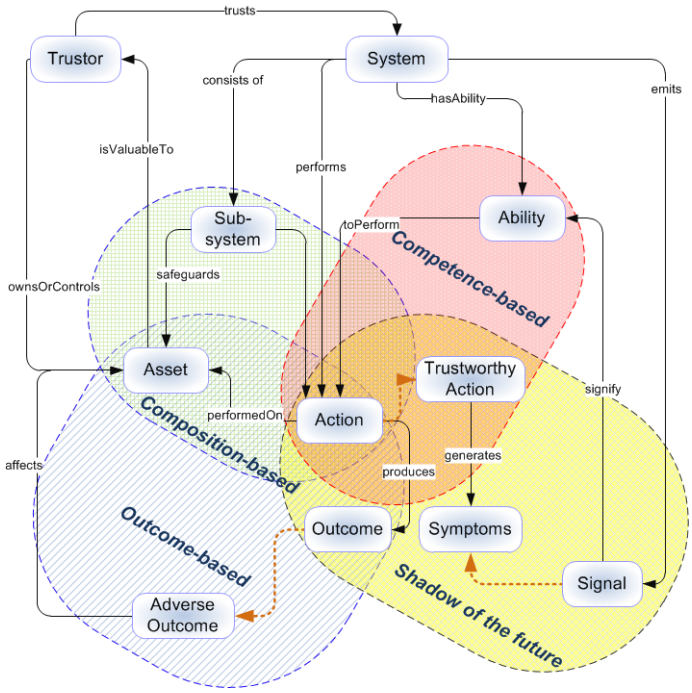


Figure 3.5: Identification of trust properties classes

against the system being relied on, whereas trust-warranting properties do not necessarily need to be checked.

3.4 Trust Property Classification

In this section, we group properties based on the concepts that each property may be concerned with. Using the trust model presented in Section 2.2.2, we identify four classes of properties, where each class is defined in terms of the concepts (from the model) that are necessary to express such properties. The classes are illustrated in Figure 3.5 and discussed below, together with examples from the scenarios discussed in Section 3.1.

3.4.1 Competence-based Properties

As discussed in Section 2.2.2, trust can only be established if a trustee has the ability to fulfil the expectations of the trustor [139]. Consider the deployment of jobs as illustrated in Figure 3.6. Each job is split into tasks, T_1, T_2 .. T_5, which are then assigned to the nodes in the infrastructure. Suppose the resource broker is assigned the responsibility of checking the compliance of the nodes in addition to scheduling the tasks. Then, to establish trust in the grid, the researcher would have to firstly

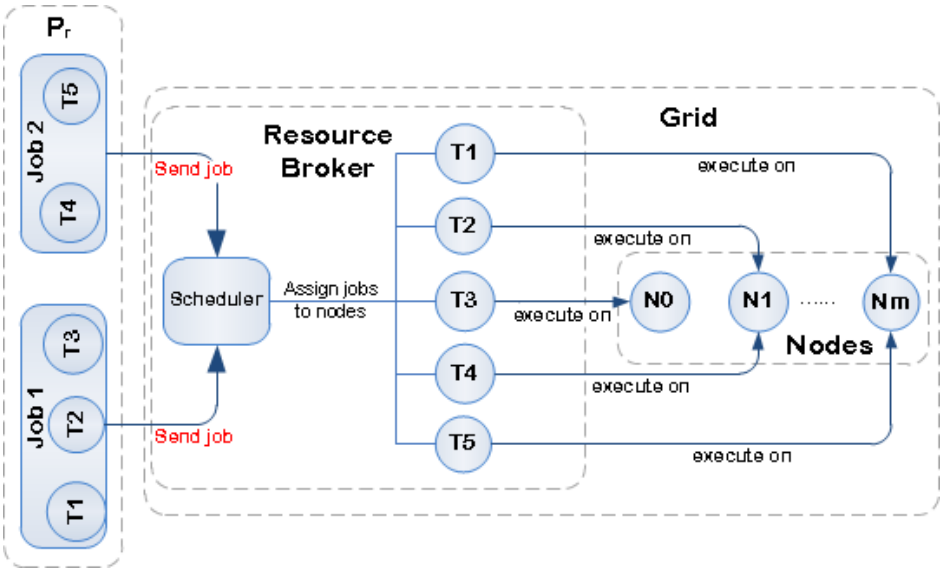


Figure 3.6: An example of a deployment of jobs onto the grid

determine whether or not the resource broker is capable of performing appropriate compliance checks on the nodes. In other words, the researcher needs to determine the competence of the resource broker to perform the checks, a requirement identified as a core attribute of trust fulfilment [109].

Using the trust model described above, competence can be captured by looking at what ability the trustee has and what actions they can perform as a result of this ability. For the relying parties, the most important concern is to determine whether the ability can lead the trustee to performing some desired actions i.e. *Trustworthy Action*. For example, in the collaborative system scenario described in Section 3.1.3, establishing trust would involve determining whether or not the combination of packages installed on a virtual machine enable it to execute the desired experiments.

We refer to the class of properties concerned with establishing the competence of a system as *competence-based properties*. These can be specified in terms of the *Ability*, *Action* and *Trustworthy Action* of the system in question. Some examples include:

- `P1`: no malicious code will be launched during a certain period, e.g. during a flicker [108] session.
- `P2`: the system prevents malicious code from using certain resources, such as network or storage.

- P3: the system ensures that only one application can access a given memory location.

3.4.2 Outcome-based Properties

Computer systems transform data from one form to another. For example, in the collaborative system, experiments may accept input in the form of name-value pairs, perform various kinds of analyses and produce output in the form of graphs and tables. In most cases, however, it is required that such transformation occurs under certain constraints. For example, the output from an experiment may be required to be generated from a specific run of an algorithm. In these cases, trust establishment would involve checking whether or not the outcome from the execution of a given set of actions is as expected. We refer to properties that are concerned with establishing the types of outcome from actions performed by the system and their impact on the behaviour of the system as *outcome-based properties*.

In the deployment scenario illustrated in Figure 3.6, the possible outcome of actions on the system can be identified by determining: i) the kinds of checks that the resource broker can perform on the nodes before assigning them with tasks; ii) what the nodes will do after receiving the tasks (e.g. can they delegate tasks to other nodes?); or iii) the level of delegation possible (i.e. will the tasks end up on a compromised node?). These properties can be captured using the conceptual model described in Section 2.2.2 to include the kinds of *Actions* that can be *performedOn* the *Assets* of interest, the *Outcome* that these *Actions produces*, and determining whether or not the *outcome* can be classified as being *adverse*.

This class includes properties that ask questions such as "will X occur or will X be prevented from occuring?" Here X is expressed in terms of some outcome possible from actions that can be performed by the system. Some examples applicable to the scenarios discussed earlier include:

- P4: sensitive data cannot flow to un-authorised devices.
- P5: security relevant components cannot be influenced by non-security sensitive parts of a system [59].
- P6: security credentials, such as private keys, can only be used when the system is in a given state [118].

3.4.3 Composition-based Properties

In the grid scenario described in Section 3.1.2, we highlighted the lack of mechanisms for providing proofs about the behaviour of such systems as one of the main challenges towards trust establishment. Consider the deployment scenario illustrated in Figure 3.6, where tasks from a single job end up on multiple grid nodes. To establish trust, a researcher would have to determine that appropriate controls are in place to ensure that: i) the resource broker behaves according to her expectations; and ii) each node that can potentially execute the tasks behaves according to her expectations. These controls would exist in the form of policy enforcement points or job manager [31].

In the collaborative scenario, the researchers would need to check the presence of certain packages, such as SELinux, on the virtual machines before they use them for their experiments. Similarly, in the password manager scenario, the user would need to check that the system uses a genuine *passman* application.

In these cases, the entities wishing to rely on the system can specify trust-properties in terms of existence, or non-existence, of components known to have some desired properties. Trust is, therefore, established if the system can securely demonstrate that all the components specified in the trust-properties exist, or do not exist on the system. This is the view taken in the TCG-based integrity verification, where integrity measurements indicate the identities of known components. This approach can be considered as a problem of establishing the composition of a system. We, therefore, refer to trust-properties that can be expressed in this manner as *composition-based properties*. Some examples of properties that would fall into this class include:

- `P7`: a job manager ensures that sensitive data is never exposed to middleware [31].
- `P8`: a configuration manager that is capable of detecting configuration changes exists on the system [72].
- `P9`: a known firewall has been installed on the system.

This class of properties can be captured using our trust model as a subset that includes both the actions and the parts or constituents of the system that performs those actions.

3.4.4 "Shadow of the Future" Properties

McKnight and Chervany [109] identify predictability in the trustee's actions as an essential prerequisite to trust. For example, in the collaborative system scenario, researchers would have to establish that the behaviour of a particular virtual machine they intend to use for their experiments conforms to expectations. This can be achieved if the virtual machine's behaviour is predictable.

In such cases, the aim would be to identify information about events observed on the system and to determine whether or not this information suggests that the system has performed or is prepared to perform some expected actions. The main idea is that if patterns of behaviour observed on a system are sufficiently similar to historic patterns that eventually led to trustworthy behaviour, then the system can be expected to follow a similar pattern, which leads to trustworthy behaviour.

Riegelsberger et al. [139] state that when a trustee performs actions, remnants, called *symptoms*, are produced. We can, therefore, identify a class of properties which are concerned with identifying the symptoms and determining whether the symptoms resulting from execution of certain actions have any patterns that are known to lead to trustworthy behaviour.

In the collaborative scenario, researchers would need to establish that the configurations of the virtual appliance can lead to trustworthy behaviour. If an experiment with similar configuration requirements was performed on the same infrastructure and the experiment was considered to have been successfully executed without compromising any trust-related constraints, then the users would benefit from this by comparing symptoms (e.g. event logs) on the virtual appliance to determine if any patterns exist that would suggest future trustworthy behaviour. Riegelsberger et al. [139] refer to this as the *shadow of the future*, in that it gives an indication of the possible future behaviour of the trustee. We, therefore, refer to the class of properties that are concerned with determining the possible future behaviour of the system as *"shadow of the future" properties*.

Such properties can be captured using the trust model to include the concepts *Action*, *Outcome*, *Symptoms*, *Signal* and *Trustworthy Action*, as illustrated in Figure 3.5. That is, given that the system *performs* certain *Actions* and *produces* certain *Outcomes*, then the *Symptoms generated* from such activities can be analysed to determine whether or not they warrant future trustworthy behaviour. Some examples of properties that fall into this class include:

- `P10`: algorithms passed to the system are the same ones that were used to produce a given output [121]
- `P11`: grid middleware, such as Condor [166] or Globus [47], when compromised, does not lead to compromises in the integrity and confidentiality of the researcher's data [31]
- `P12`: new credentials will be generated when the system runs.

3.5 Architectural Aspects

In the previous section, we grouped trust-properties into four classes, where each class includes properties that are framed using similar concepts. In this section, we make use of the architectural model discussed in Section 2.1.2 to identify aspects of a system that may be useful for reasoning about each class of properties.

Based on a translation of the *Vitruvian triad*[1], Albin [6] defines three aspects of an architecture that correspond to the three principles in the Vitruvian triad. These include *form*, *function* and *fabrication*, so that a system is constructed using a certain form in order to satisfy the function and both are realised through the process of fabrication. We base our architectural aspects on these three. In the following sections we discuss each of these aspects and draw out a subset of the architectural model necessary to describe each aspect.

3.5.1 Structural Aspect

Form, also referred to as structure, corresponds to *venustas*, which is concerned with the quality or attractiveness resulting from the use of well developed elements [6]. It defines the quality of elements used and how their interconnections help to create a balance among various competing forces, such as performance versus security.

When coming up with an architecture, an architect considers various structures that can be used to construct the system and then determines which of the structures assumes a good balance between all competing attributes. This means that the structure embodies information about how the choice of elements and their combination helps to induce a system's properties. For this reason, the structure can be used as a good source of proofs about the behaviour of a system.

[1]The Virtuvian triad defines principles that an architecture should exhibit and include firmitas, utilitas and venustas.

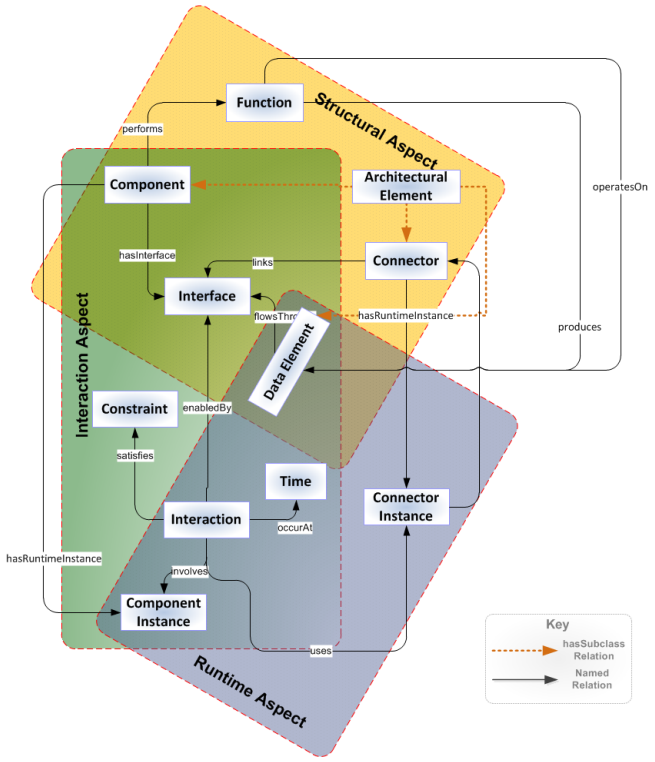


Figure 3.7: Identification of architectural aspects (represented by an ellipse)

Using the architecture model, the structure can be captured by considering the architectural elements that will be used in a system and the dependencies or inter-relations among them. Garlan [154] refers to these as structural characteristics of the configurations. We, therefore, define the structural aspects as a subset of the architectural model (comprising of *Component*, *Connector*, *Function*, *Data Element* and *Interface*) which can be used to infer concerns about the quality resulting from the combination of certain architectural elements.

In the deployment scenario illustrated in Figure 3.6, the structure would include components such as resource broker, scheduler and compute nodes and connection mechanisms such as *http*, *ssh*, *rpc*, etc. In addition, the functions of each of the components would be specified together with interfaces through which each component can interact with other components.

3.5.2 Runtime Aspect

Albin [6] states that quality in the product of an architecture (i.e. the system) is determined not only by the design but also the technologies used in its implementation. Garlan [154] also notes the importance of component choice by stating that

extra-functional properties system components play an important role in the quality of the system.

The *firmitas* principle, which is concerned with the quality of the materials used in construction of the system, can be used to understand how the choice of architectural elements influences the properties of a system. This principle is captured as *fabrication* in Albin's model, referring to both the process of developing the system and the result of this process (i.e. running system).

In our architectural model, the product of an architecture is symbolised by the concepts that exist at *runtime*. That is, the product corresponds to the instantiation of the architecture into a system [141]. In this case fabrication refers to both the instantiation process and the resulting system. At runtime, the system will comprise of component and connector instances as well as data, as shown in Figure 3.7. The types of instances that exist at a given time determines the available interfaces through which data can flow between existing instances as well as to external entities. We therefore define the *runtime aspects* as a sub-model that includes *Component Instance*, *Connector Instance*, *Data Element*, *Interaction* and *Time*. *Component Instance* is used to specify the computational elements that are available to process data while *Connector Instance* is used to denote the channels that are available for data to flow through.

Griffin et al. [61] identify three main categories of attestation properties. These include environment, data-related or processing. These can be mapped to runtime, structure and interaction aspects respectively.

3.5.3 Interaction Aspect

Albin [6] defines *System Function* as the need or use of the system or the services that the system is meant to provide and points out that it corresponds to *utilitas* in the Vitruvian Triad. It follows from the *form* of the system and is therefore determined by the kinds of activities that are enabled by the constituent elements. For example, in the collaborative scenario, two virtual appliances created through the same process may offer different services depending on what packages (constituents) are installed on each. Similarly, two identical constructed virtual appliances may still offer different services if they are configured differently. For example, if both virtual appliances have a firewall installed, they may still allow different traffic depending on the configuration of the firewall, or whether or not the firewall is started in the first place.

This illustrates that not only is the composition or structure of the system important, but also how the constituents interact as a result of the configurations of the system. Therefore, to determine the behaviour of a system, one needs to look at all the possible interactions that can be created by system components as well as the kinds of data that can flow to the interfaces involved in these interactions.

In the architecture model described above, an interaction is defined as a concept that relates instances of architectural elements involved in communication or coordination together with interfaces through which this communication or coordination occurs. We therefore define the *interactions aspect* as a sub-model that determines the *System Function*. It comprises of *Component*, *Interaction*, *Constraint*, *Interface* and *Time*, as shown in Figure 3.7. *Component* is used to specify the architectural elements involved in a particular *Interaction*. *Constraint* is used to capture the condition that must be satisfied before an interaction can occur while *Time* is used to determine the order in which the interactions occur.

3.6 Mapping Property Classifications to Architectural Aspects

In Section 3.4, we identified four classes of trust properties that can be determined of a system. In the previous section, we identified sub-models of a system architecture that focus on different aspects of a system. Motivated by the idea that a system architecture is vital for the system's trustworthiness [16], in this section we investigate how the identified architectural aspects could be used to reason about properties in each class.

The idea is that the process of architecture design involves identifying the best balance between various competing attributes to ensure that the system will exhibit desired properties. This means that the composition and structure of the system together with the environment in which the system operates will determine the intrinsic and contextual properties of the system [139], referred to as trust-warranting properties by Bacharach and Gambetta [12]. On the other hand, trustors (users and other systems) have constraints that they expect the system to satisfy, which we call trust-properties. Therefore, establishing trust in a system is essentially a process that involves mapping the trustors' expectations to the guarantees that a system provides. For this reason, we identify the possible mappings between the architectural aspects and the trust-property classes.

3.6.1 Competence- and Composition-based to Structural Aspect

The design of an architecture begins with the definition of the purpose for which the system is required as well as constraints under which the system should fulfill its purpose. Assuming that the design is adequate, the system will be able to provide services for which it is intended. However, even when two systems have been designed to provide identical services, one may opt to use one system over the other. The reason for this is that several decisions are made during the design to improve the way the system responds to changes in the environment, for example how it scales with rising demand for its services. These decisions may result in the introduction of new components or changes in the placement of components.

In Section 3.5.1, we discussed the relationship between the structure or form of a system and its quality or attractiveness. The structure of the system therefore determines both the competence — what the system can or cannot do — and the composition — what components are used and how these work together to provide the services the system is intended for. This realisation enables us to create links between the competence- and composition-based properties to the structural aspects of a system.

Consider the property `P1`, *no malicious code will be launched during a certain period* from the competence-based class and `P8`, *a configuration manager detects configuration changes on the system* from composition-based class. To reason about `P1` would require checking that: i) the system is capable of launching malicious code; ii) the system has mechanisms for checking which code is malicious; and iii) mechanisms are in place to control what can be launched and what can't. To reason about `P8` would require checking that: iv) the system is capable of including a configuration manager as part of its architecture; and v) system has configurations which may be changed.

Condition *i* can be checked by looking at the composition of the system to identify if there are any components that handle the execution other code. Similarly, condition *iv* would involve checking the composition to determine if there is any way that a configuration manager could be included. This can be achieved through the architectural elements defined in the structural aspects creating a link between the sub-systems defined in the composition-based model to components in the structural model, *Link 3* in Figure 3.8.

Conditions *ii* and *iii* are concerned with capabilities of the system. Condition *ii* is about the ability to distinguish code that executes on the system while condition

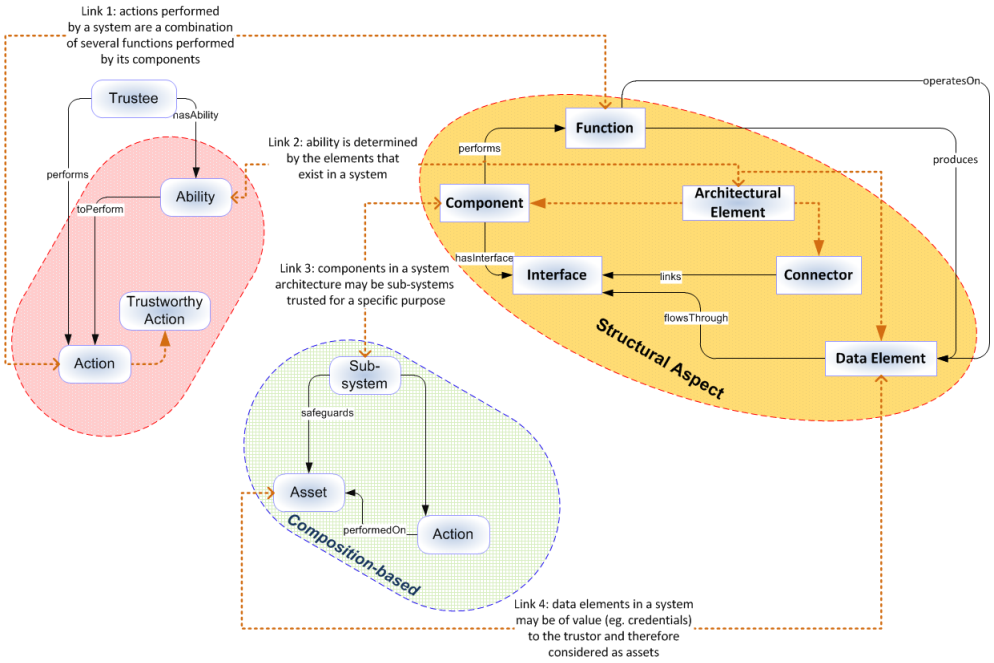


Figure 3.8: The mapping between the competence- and composition-based properties to structural aspects

iii is about controlling what gets launched. Both require checking the ability brought about by the choice of architectural elements (i.e. choosing an intrusion detection component versus quarantine or repair mechanisms). Therefore, the ability defined in the competence-based model is related to architectural elements, *Link 2* in Figure 3.8. By realising that actions on a system are combinations of several functions performed by its components, we can relate an action from the competence-based model to *Function* defined in the structural aspects, *Link 1* in Figure 3.8.

Condition *v* would involve checking the composition of the system to identify any data that is used to change the settings of the system. These configurations may also be sensitive and could thus be considered an asset, *Link 4* in Figure 3.8.

This means that the components that exist within the system architecture and how these components are related can be used to reason about the composition and competence of a system. The components are sub-systems which perform functions that contribute to the overall actions that the system can perform.

3.6.2 Outcome-based to Interactions Aspect

The architectural model includes the concept of an *Interaction* to capture the idea that functional elements can interact in order to fulfil some functionality, such as

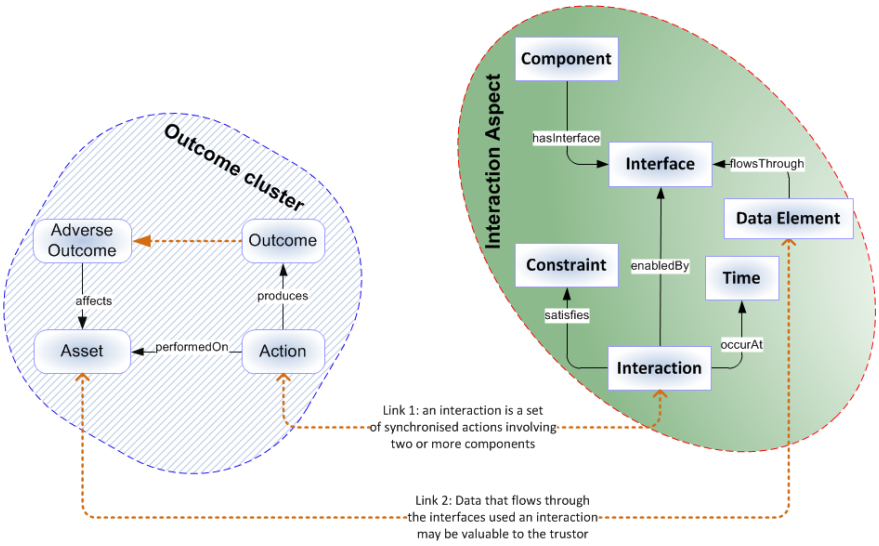


Figure 3.9: The mapping between the outcome-based properties and interactions aspect

service provision. This idea can be captured by realising that an *Interaction* can be represented as a set of actions involving two or more components. It is distinguished from a *function* which can be thought of as a type of *Action* performed solely by a single entity, i.e. a component. These interactions result in new data and/or new functionality and can, therefore, be used to reason about properties that involve looking at the possibility of certain kinds of outcome occurring or not occurring, i.e. outcome-based properties.

Consider an example property, `P5`, *security relevant components cannot be influenced by non-security sensitive parts of a system.* To reason about this property, one needs to determine a few things about the system including: i) distinguish security relevant components from non-security relevant ones; ii) determine if there are possible instances when any two components, one from each set, interact; and iii) determine the nature of interactions, if any, to identify whether or not there exists some undesirable outcome, i.e. adverse outcome.

Distinguishing between security relevant and non-security relevant components can be achieved by first realising that these two would be defined as components of the system. To determine that condition *ii* was the case, the components identified (i.e. both security and non-security relevant) would need to be analysed to determine whether they have interfaces that would allow interaction with other components. This is achieved through the relationship between an interaction and an interface, as defined in the interactions aspect. Once the interfaces are available, the components

perform certain kinds of actions, such as initialise a communication link or send data, to create an interaction. For this reason, an interaction can be identified in the same way as in the previous section, i.e. a set of synchronised actions. This relationship is represented as *Link 1* in Figure 3.9.

Condition *iii* can be checked by observing the impact of certain data, of interest, flowing to the interfaces that are used in the interactions. This data can be considered of value to the trustor and therefore to establish trust the trustor would need to look at the outcome of such information flows. We can therefore, relate assets to data that may flow on the interfaces, as illustrated by *Link 2* in Figure 3.9.

This means that by looking at the interactions within the system, we can reason about the possible outcomes such as "can data flow through a given interface?" or "can some given components interact?", and determine their adversity.

3.6.3 "Shadow of the Future" to Runtime Aspect

In Section 3.4.4 we defined *"shadow of the future"* as a class of properties concerned with establishing the possible future state of a system. This state is determined by looking at the *symptoms* generated from actions performed on the system. Consider the property, `P12`: *algorithms passed to the system are the same ones that were used to produce a given output.* To reason about this property, we would need to check for a number of things, including: i) there were mechanisms in place to accept the algorithms; ii) algorithms were actually passed to the system in question; and iii) there was some output observed from the system at some point in time.

To determine that condition *i* was the case would require checking the existence of runtime artifacts that can be used to accept input (i.e. algorithms). These would be in the form of connector and component instances, defined in the runtime aspects. Component and connector instances are a result of some activity on the system such as instantiation. We can, therefore, link component and connector instances as outcomes, as illustrated in *Link 1* in Figure 3.10.

Similarly, to determine that condition *ii* was satisfied would require checking for the presence of interactions that make use of the component and connector instances capable of accepting the algorithms. The time that such interactions occurred can also be checked to provide further information about the sequence of interactions. Interactions can be considered as synchronised actions performed by two or more components of the system. For this reason, we map the concept of an action defined in the *"shadow of the future"* to an interaction as illustrated in *Link 2* in Figure 3.10.

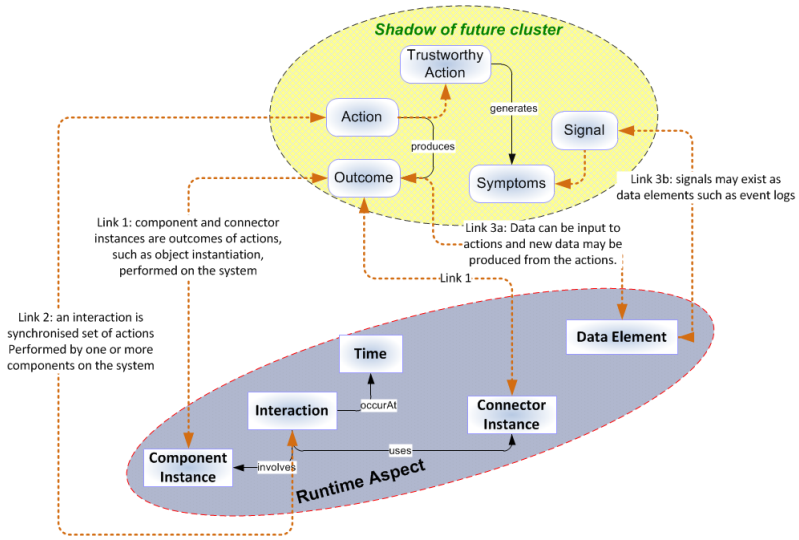


Figure 3.10: The mapping between the shadow-of-the-future and runtime aspects

Finally condition *iii* can be determined by checking that some data was generated as a result of the interactions created by the component and connector instances. This can be achieved by checking the relationships of the data elements on the system to system elements (e.g. was the data generated by a given component?) Furthermore, the symptoms defined as part of the *"shadow of the future"* can be considered as some form of data elements (e.g. event logs), which is also included as part of the runtime aspects. Link 3 in Figure 3.10 denotes the relationship between data element defined in the runtime aspect to symptoms and outcome defined in the *"shadow of the future"* model.

This mapping demonstrates that the properties included in the *shadow of the future* class can be determined by looking at the runtime aspect of the architecture., in which case, symptoms will exist in the form of available instances (created from the action of instantiation) or data such as event logs or provenance data. This creates the links between the shadow of the future and the runtime aspect as illustrated in Figure 3.10.

3.7 Towards Verification Models

Having established the link between the system architecture and its trust properties, how can we reason about these properties? To address this question, we begin by making a few notes.

First, due to the complexity of modern system, a single representation of the

system would be too complex to enable reasoning about all kinds of properties. Second, any given system will have a number of stakeholders each of which may have a restricted view of the system as a result of their knowledge about the system or limitations on the amount of information available (e.g. as a way of balancing other attributes such as privacy). Finally, a system architecture embodies information about the nature of its elements and relationships among them [13].

Therefore, to reason about trust-properties and support multiple perspectives while dealing with complex systems requires suitable representations of knowledge about the system. This requirement can be satisfied by using architectural models, where a model is defined as an approximation or idealisation of the selected aspects of a system which represent knowledge about the system sufficient to reason about its properties [75].

The next three chapters will be dedicated to the development of these models and frameworks that guide their usage. But before we do so, we identify the key requirements that have to be satisfied by each model. These requirements were elicited through discussions held with system administrators and managers at OeRC as well as giving consideration to trusted computing literature.

3.7.1 Requirements for a Model to Capture Structural Aspects

The structural aspect is concerned with establishing the composition of the system and, therefore, any model that is intended to cater for the structural aspect must be capable of handling the multitude of constituents of systems. More specifically, the model must:

`REQ 1` Identify and characterise the building blocks of trustworthy system behaviour.

`REQ 2` Enable tracing of the overall properties to the constituents of the system.

`REQ 3` Support re-use of system constituents in multiple application domains.

`REQ 4` Provide mechanism for reasoning about properties resulting from the use of a specific structure.

3.7.2 Requirements for a Model to Capture Interaction Aspects

Interactions are defined as a synchronised set of actions performed by two or more components on a system. This synchronisation occurs over the interfaces provided

by the components. The choice of a model for representing interaction aspect of a system must therefore be capable of:

`REQ 5` Capturing both the components involved in the interaction as well as channels through which these interactions occur.

`REQ 6` Supporting concurrent execution of system components.

`REQ 7` Supporting the construction of system instances with varying properties.

`REQ 8` Providing information describing the reasons for failing to satisfy a given property so as to allow the relying entity to determine further course of action such as remediation.

`REQ 9` Providing mechanisms for exploring all the possible interactions that may occur on a system.

3.7.3 Requirements for a Model to Capture Runtime Aspects

The runtime aspect is concerned with manifestations of architectural instances. Therefore, a suitable model to capture runtime aspects must be capable of generating a system representation from runtime information. More specifically, the model must:

`REQ 10` Support collection of information that may serve as reliable attestations [19] and as incentives for trustworthy action [139].

`REQ 11` The information referred to in `REQ 10` should be a true representation of the actual events that occurred on a particular system.

`REQ 12` The information described in `REQ 10` should be made available to subsequent entities that intend to rely on the system.

`REQ 13` Represent information referred to in `REQ 10` in a format that allows the relying entity to reason about fulfilment of trustworthy actions.

3.8 Summary

The challenge of establishing trust in a system can be attributed to a number of factors including complexity (resulting from increased expectations on systems), the lack of consensus on what it means to trust a system, and, a lack of a systematic approach that can be used to reason about the impact of using certain technical mechanisms on trust. This chapter proposed a framework that uses an architectural approach as a means of managing the complexity through the provision of abstractions. The abstractions provided by this approach also serve as a means for reasoning about certain classes of trust-properties. The framework also motivates the need for and guides the development of models that represent knowledge about the system in a manner suitable for reasoning about certain classes of trust-properties.

In the next three chapters, we develop models that are designed to meet the requirements discussed in the previous section and are suitable for representing the system for the purpose of evaluating trust-properties that fall in a particular class.

Chapter 4

Semantics of Trustworthy Systems

In Section 2.3, we discussed how trustworthy systems make use of existing security mechanisms such as cryptography, protected storage and remote attestation, to enable trustworthy operation. Several systems [30, 94, 155, 149, 71, 52] use these techniques to solve problems in specific application domains. However, not much attention has been directed towards providing a systematic approach that can be used to justify the basis of trust in such systems. As a result, it is still not clear how the use of these techniques influence trust-properties or why one system might be considered more trustworthy than another despite both systems using the same techniques. For example, by using a dynamic root of trust rather than a static one, how does that change the properties of a system? What assumptions are necessary when such mechanisms are used?

In this chapter, we demonstrate that a systematic characterisation of the building blocks is essential for reasoning about how the structure of the system affects competence and composition related trust-properties. Based on the requirements identified in Section 3.7.1, we build such a characterisation in the form of a *semantic model*, which defines the fundamental concepts used in trustworthy systems together with relationships among them. The model formalises the concepts identified in a system architecture in Section 2.1.2 and complements these with concepts from trusted computing, which we call *trust enablers*, to demonstrate how they lead to trustworthy operation.

We begin with a discussion on the need for a semantic model and the approach adopted to build one. In Section 4.2 we present the proposed model, including its components and how they fit together to create various kinds of systems. We discuss how the resulting model can be used to reason about trust-properties in Section 4.4 followed by an evaluation of the merits and challenges of the approach in Section 4.5. The conclusion is presented in Section 4.6.

4.1 Overview

4.1.1 The Need for a Semantic Model

Consider the password manager scenario described in Section 3.1.1. Suppose the user had a choice of two platforms, M_1 and M_2, such that M_1 is based on a regular system architecture while M_2 is based on a trustworthy system architecture.

As illustrated in Figure 4.1, M_1 loads the components on the system without recording their integrity whereas M_2 measures every component before executing it. A boot loader is loaded and executed, which loads an operating system. The operating system then loads and executes both *passman* and *passcracker*.

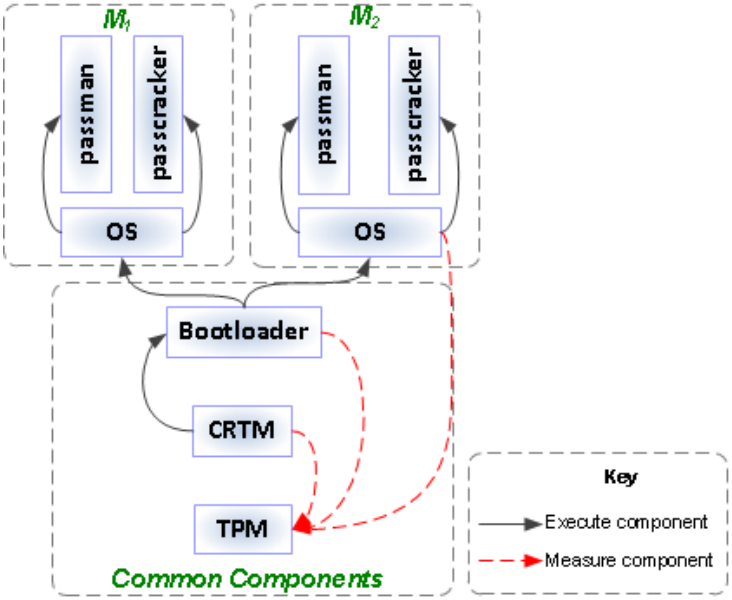


Figure 4.1: Example illustrating the boot processes of three different platforms

In this scenario, the ability of *passcracker* to deduce the password generated by *passman* depends on two main factors. First, the architecture of both applications, *passman* and *passcracker*, determine the capability that each application has. For example, if *passman* does not manage its own resources, such as database connections or sessions, properly, then *passcracker* could get hold of the password by compromising the weaknesses in the use of resources (e.g. re-use an unclosed connection). Second, the architecture of the system on which these applications run determines the kinds of interactions, as discussed in Section 3.5.3, that can occur on the system at any given time and, thus, could be used to implement protection mechanisms. For example, a protection mechanism might be implemented by enforcing that no interactions with a memory device are allowed while *passman* is running a certain process.

We note that an architecture embodies information about the components (or building blocks), relationships among them and principles governing the design and evolution of a system [76, 13]. However, this information on its own is not sufficient to enable the kind of reasoning desired by the user in this particular scenario. For example, the system architecture of M_2 ensures that every component can be identified. However, this information is not very helpful unless the user already understands the characteristics of each component that makes its presence valuable. This problem is related to that identified of TCG-based remote attestation [146, 64] in that remote attestation helps to identify the presence of components; however, it does not provide a systematic approach for reasoning about how the component's presence results in certain behaviours. In order to enable such reasoning, the components in a system must be characterised to understand their properties and how they can be used together towards a common goal, e.g. protecting the passwords generated by *passman*. We describe such a characterisation in Section 4.2.

4.1.2 Approach and Choice of Model Representation

In the scenario discussed above, the entities such as boot loader, operating system and applications can be considered as elements in the system's structure. Each element provides services to other elements through its interfaces and makes use of functionality provided by other elements. Furthermore, each element has state, which is determined by the data it has access to and the state of its interfaces, as well as the computations it is able to perform. Therefore, the choice of the model must be capable of capturing both the state of the elements as well as means for changing the state.

One possible option would be to use architecture description languages (ADL) such as WRIGHT [141] or RAPIDE [96] or modelling frameworks such as UML statecharts [14]. Using an ADL would require that the system be represented in the semantics of a particular ADL. The problem, however, is that each of these ADLs is specifically designed for certain kinds of systems and therefore the semantics are only adequate when dealing with those systems for which the ADL is best suited. And indeed significant effort continues to be directed towards designing ADLs that are domain independent.

Our aim, however, is not to contribute to the significant knowledge in ADL design. Instead, we aim to be ADL neutral and define the semantics in a way that can be translated into any specific ADL if needed. For this reason, we adopt a simple framework [81] that is based on the Z language [159]. We follow the structure proposed

by Woodcock [177] and Jackson [80] so that the schema language of Z is used to model elements of a system as abstract types and operations upon these types. Here a schema is defined as a signature that expresses the constituents of an element together with the properties over the signature [159]. We use the schema calculus to combine schemas to make more complex types. For example, we can define the elements such as BIOS, boot loader, OS kernel and applications as instances of a *Component* schema type and then combine the instances using schema calculus to create a complete system. We then use predicate logic to define assumptions made in a system and properties that hold when these assumptions are true.

4.2 A Semantic Model

In this section we build a *semantic model* that uses the generic system model, described in Section 2.1.2, as a foundation and complements it with building blocks, which we refer to as *trust enablers*, that provide the core trusted computing primitives including attestation, secure storage and integrity measurement. This model allows us to reason about trust-properties of a system that is constructed using these building blocks.

The semantics of the main Z constructs used in the model appear in Appendix B. The model discussed in this section has been simplified for legibility and is not meant to be complete (e.g. some of the operations referred to are not presented here) or syntactically perfect. Instead, we provide the full model in Appendix C.

4.2.1 Capturing the System Architecture Model

A model of a system architecture was informally discussed in Section 2.1.2. To enable reasoning about trust-properties, in this section we describe the formal specification of this model to comprise of *Data*, *Interfaces*, *Components*, *Connectors* and *Interactions*.

Data and Interfaces

Data is an important constituent of any system. Every system handles several kinds of data, some of which remain relatively static, such as credentials, while other types change frequently, such as software code — as a result of updates and bug fixes. For example, in the grid scenario described in Section 3.1.2, data such as experiment executable code is moved from a researcher's platform to a resource broker, which sends the data to a node capable of executing the job. To achieve this, each component, such as resource brokers, researcher platform and grid node, exposes interfaces

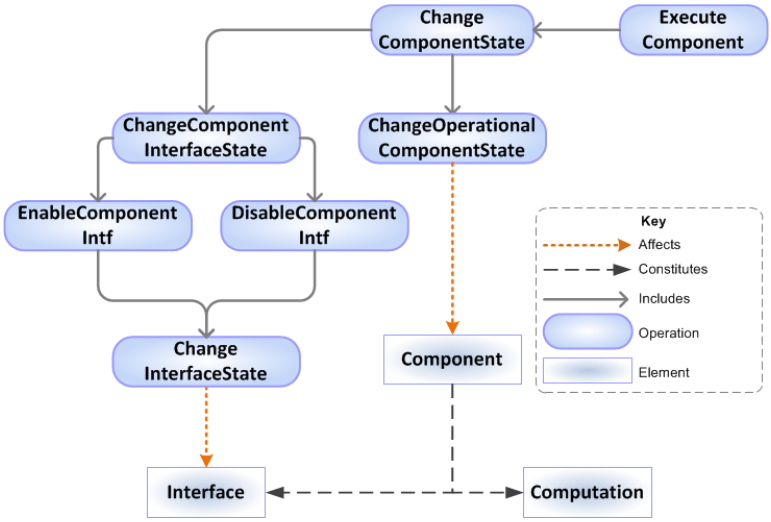


Figure 4.2: An illustration of the structure of a component in the model

through which data can flow into or out of the component. We therefore define *DATA* as a basic type and *Interface* as an abstract type that has state, *intfState*, which may be *enabled* or *disabled*. This state can be changed using the *ChangeInterfaceState* operation, which simply applies the new state if it is different from the current state.

$$[DATA, ADDRESS]$$
$$INTERFACESTATE ::= enabled \mid disabled$$

Interface

$$intfState : INTERFACESTATE$$

Components

Components are processing units in an architecture [130]. For this reason, we specify components to include a computation aspect that represents their processing capabilities. We model this functionality as an abstract data type *Computation*, which comprises of a set of interfaces through which the functionality can be triggered and a piece of *logic*, representing the actions that it performs to provide particular functionality. The entire functionality of the component is therefore represented as a set of computations, *cmpComputation*. In order for this functionality to be accessible to other components, each component offers one or more interfaces, *cmpInterface*, such that all the trigger interfaces defined for its computations exist within this set.

Figure 4.2 illustrates how parts of the specification are combined to create the structure of a component. A component has an operational state, *opState*, which may

take up any of the values *notloaded*, *loaded* or *running* such that only components that have been *loaded* can be executed.

$[NAME, LOGIC]$
$COMPONENTSTATE ::= running \mid loaded \mid notloaded$

Computation

$triggerInterface : \mathbb{P}\, Interface$
$logic : LOGIC$

We further specify that each component has at most one interface, *execInterface* through which it can be executed. This could be a *main* method in Java applications. This interface is expected to be *enabled* before a particular component can be executed. This enables us to capture executable, e.g. applications, and non-executable components such as shared libraries, so that non-executable components are captured as components which have an *execInterface* that is always disabled.

Component

$opState : COMPONENTSTATE$
$execInterface, measureInterface : NAME$
$cmpInterface : NAME \pfun Interface$
$activeCMPInterface : \mathbb{P}\, NAME$
$cmpComputation : NAME \pfun Computation$

$measureInterface \neq execInterface$

$execInterface \in (\mathrm{dom}\ cmpInterface)$

$measureInterface \in (\mathrm{dom}\ cmpInterface)$

$activeCMPInterface = \{n : NAME \mid (cmpInterface\ n).intfState = enabled \bullet n\}$

$\forall c : (\mathrm{ran}\ cmpComputation) \bullet c.triggerInterface \subseteq (\mathrm{ran}\ cmpInterface)$

InitComponent

$Component$

$\forall Interface \mid \theta Interface \in (\mathrm{ran}\ cmpInterface) \bullet InitInterface$
.
.
.

PromoteChangeCMPInterfaceState

$\Delta Component$
$\Delta Interface$
$intfName? : NAME$

$intfName? \in (\text{dom } cmpInterface)$

$cmpInterface' = cmpInterface \oplus \{intfName? \mapsto \theta Interface'\}$
.
.
.

$ChangeComponentIntfState \mathrel{\widehat=} \exists\, Interface \bullet$
$ChangeInterfaceState \land PromoteChangeCMPInterfaceState$

When a component is initialised, all its interfaces are *disabled*. This can later be changed using the *ChangeComponentIntfState* operation, which is a promotion of the *ChangeInterfaceState* operation defined on the *Interface* schema. We define a number of operations on the component to represent specific changes in the state of a component including *LoadComponent*, *EnableComponentExecution* and *ExecuteComponent*.

EnableComponentExecution
ChangeComponentIntfState

$opState = loaded$
$execInterface \notin activeCMPInterface$
$newIntfState? = enabled$
$intfName? = execInterface$

LoadComponent
ChangeComponentInternalState

$opState = notloaded$
$newCMPState? = loaded$

EnableComponentMeasurement
ChangeComponentIntfState

$opState = loaded$
$measureInterface \notin activeCMPInterface$
$newIntfState? = enabled$
$intfName? = measureInterface$

ExecuteComponent
ChangeComponentInternalState

$opState = loaded$
$execInterface \in activeCMPInterface$
$newCMPState? = running$

Connectors

Similar to the model adopted in WRIGHT [141] and ACME [53], we specify a connector as a combination of a set of interfaces, referred to as roles in WRIGHT, through

which components can interact and a protocol used to coordinate the interactions. At any point in time, the interfaces available will be determined by the protocols currently executing. To capture this, we specify *activeCNTInterface* as a subset of the interfaces available on a connector that only includes those that are active at any given point.

Since connectors are not computational entities, we can focus on the state changes of their interfaces for the purpose of understanding how they behave. These are captured in the *ChangeConnectorState* operation which can either enable or disable interfaces on the connector.

Connector

$$cntProtocol : NAME \pfun PROTOCOL$$
$$cntInterface : NAME \pfun Interface$$
$$activeCNTInterface : \mathbb{P}\, NAME$$

$$activeCNTInterface = \{n : NAME;\ i : Interface \mid i = cntInterface\, n \land i.intfState = enabled \bullet n\}$$

InitConnector

$$Connector$$

$$\forall\, Interface \mid \theta Interface \in (\operatorname{ran} cntInterface) \bullet InitInterface$$
.
.
.

$$ChangeConnectorState \mathrel{\widehat=} EnableConnectorInterface \lor DisableConnectorInterface$$

Interactions

Components on a system interact with each other in order to exchange data or work together on a task. These interactions occur at a specific time and involve at least two components. We specify interactions to include the components involved together with the interfaces they use and the interfaces on the connector to which they connect. All interfaces, both on the connector and component, that are used in an interaction are required to be *enabled*.

Interaction

$$occursAt : Time;\ cntInstance : NAME$$
$$cmpIntfInstance : NAME \times Interface \pinj Interface$$

$$\forall\, x : NAME;\ i, j : Interface \mid j = cmpIntfInstance\,(x, i) \bullet i.intfState = enabled \land j.intfState = enabled$$

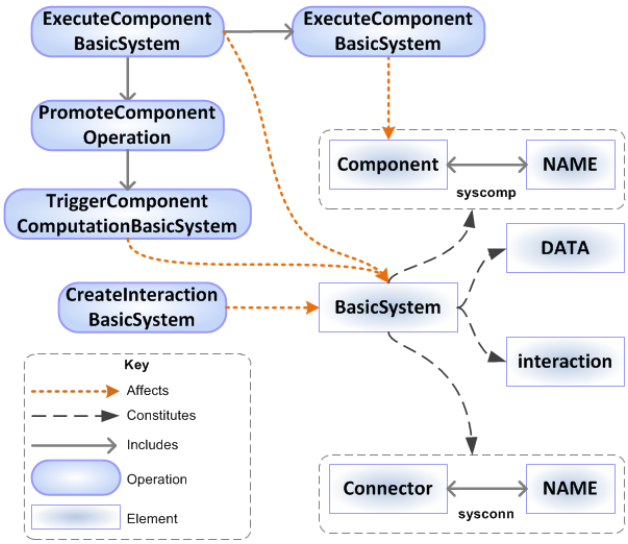


Figure 4.3: Constituents of a basic system and operations on them

Interactions are created at the system level, so we defer the definition of operations to create interactions until we discuss the structure of a basic system in the next section.

Basic System

So far we have formalised various constituents of a system architecture. We now discuss how these are combined into a system. We define a system as a composition of components, defined as a function *syscomp*, and connectors, defined as another function, *sysconn*, that creates interactions based on the state of the interfaces available on the components and connectors. The constituents of a basic system and how operations on them relate to system state are illustrated in Figure 4.3.

We define a basic system to include state, *live* or *offline*. This state determines what activities can occur on the system, so that computations and interactions can only occur on a *live* system. For this reason, system initialisation ensures that the set of interactions on the system is initially empty. The system state can be changed using the operation *ChangeSystemOperationalState*, which applies the new state if it is different from the current state.

BasicSystem

$$
\begin{array}{l}
interaction : \mathbb{P}\, Interaction \\
sysconn : NAME \pfun Connector \\
syscomp : NAME \pfun Component \\
activeComponent : \mathbb{P}\, NAME \\
data : \mathbb{P}\, DATA;\ sysState : SYSTEMSTATE \\
\hline
activeComponent = \{n : \mathrm{dom}\, syscomp \mid (syscomp\ n).opState = running\} \\
\forall\, x : interaction \bullet \\
\quad x.cntInstance \in \mathrm{dom}\, sysconn \land \\
\quad \mathrm{dom}(\mathrm{dom}\, x.cmpIntfInstance) \subseteq \mathrm{dom}\, syscomp
\end{array}
$$

InitBasicSystem

$$
\begin{array}{l}
BasicSystem \\
\hline
sysState = offline \\
\forall\, Component \mid \theta Component \in \mathrm{ran}\, syscomp \bullet InitComponent \\
. \\
. \\
.
\end{array}
$$

BasicSystem operations, such as *StartBasicSystem*, *LoadComponentBasicSystem* and *CreateInteractionBasicSystem*, change the state of the system or that of its constituents. The overall change in the system's state comprise of one or more of the following:

- a change in the state of the connectors or components that exist on the system;
- a change in the operational state of the system (*live* to *offline* or vice-versa); or
- creation of new interactions.

Interactions are created using the *CreateInteractionBasicSystem*, which requires that all the interfaces provided must have been *enabled*, the components involved must be *running* and the system must be *live*. This operation results in the change of the *interaction* set on the system.

CreateInteractionBasicSystem

$\Delta BasicSystem$
$cmpName1?, cmpName2?, connName? : NAME$
$cmpIntf1?, cmpIntf2?, cntIntf? : Interface$
$occursAt? : Time$

$sysState = live$

$cmpName1? \in activeComponent$

$interaction' = interaction \cup \{i : Interaction \mid$
$\quad i.occursAt = occursAt? \land i.cntInstance = connName? \land$
$\quad i.cmpIntfInstance = \{(cmpName1?, cmpIntf1?) \mapsto cntIntf?,$
$\quad\quad (cmpName2?, cmpIntf2?) \mapsto cntIntf?\}\}$

.
.
.

.

Performing Computations

Earlier we defined computations as functionality provided by components on a system. This functionality is reflected through changes in the state of the system and that of its constituents. In this view, we abstract away from the details of a component's functionality and instead focus on how the presence of a particular component, in a particular state, affects the state of other elements on the system.

We define *TriggerComponentComputation* as an operation that is performed on a particular component to initiate a computation. This is achieved through one of the interfaces defined on the computation and only when that interface is active.

PromoteTriggerComputation

$\Delta BasicSystem$
$\Delta Component$
$cmpName? : NAME$

$sysState = live$

$\theta Component = syscomp\ cmpName?$

$syscomp' = syscomp \oplus$
$\quad \{cmpName? \mapsto \theta Component'\}$

.
.
.

TriggerComponentComputation

$\Delta Component$
$trigIntf? : NAME$

$opState = running$

$trigIntf? \in activeCMPInterface$

.
.
.

Performing a computation could also affect the global state of the system. For example, new interactions could be created or connector interface states could be changed. To capture this we define *PromoteComponentComputation* to relate the change of state at the system level to that at the component level. We combine this schema with *TriggerComponentComputation* to create *TriggerComponentComputationBasicSystem* which specifies that triggering a computation could result in a change of the system's state.

$$TriggerComponentComputationBasicSystem \mathrel{\widehat=} \exists\, Component \bullet \\ PromoteTriggerComputation \land TriggerComponentComputation$$

4.2.2 Protected Basic System

Components on a system expose a number of interfaces that can be used by other components. However, in practice, not all components can use any available interface on a system. This is achieved through a number of *protection mechanisms*, which restrict access to, or usage of, the interfaces.

In this section, we identify and formalise various kinds of protections that can be applied to the *BasicSystem*. We use a schema *ProtectionSystem* to denote a protection mechanism on the system, which includes a protection function, X, that maps a particular component to a set of components from which it is protected and a component, P, that enforces the protection. The protection function is such that for any tuple (A, B) where A is drawn from the set of components and B is a set of components, A should not exist in B. In other words, it is meaningless to say that a component is protected from itself.

ProtectionSystem

$$BasicSystem$$
$$P : Component$$
$$X : Component \pfun \mathbb{P}\, Component$$

$$P \in \operatorname{ran} syscomp$$
$$\forall\, A : Component;\ B : \mathbb{P}\, Component \bullet (A, B) \in X \Rightarrow \\ A \notin B \land A \in \operatorname{ran} syscomp \land B \subset \operatorname{ran} syscomp$$

Protection through restricted interactions

Components on a system can be protected by restricting the kinds of interactions they can create among themselves. This kind of protection can be achieved through mechanisms such as network segmentation — where components protected from each

other would exist in different network segments, or virtualisation — where components in one virtual machine cannot interact directly with components in another virtual machine.

We use the *RestrictedInteractionProtectionSystem* schema to capture this kind of protection. The schema defines a protection function *restrictedInteraction* which contains all those components from the system (enforced by invariants on the generic function X defined in *ProtectionSystem*) that do not participate in the same interaction, with a given component, at any point in time — where participation in this context implies that the component exists in the *cmpIntfInstance* part of an interaction.

RestrictedInteractionProtectionSystem

$$
\begin{array}{l}
BasicSystem \\
protector : NAME \\
restrictedInteraction : NAME \pfun \mathbb{P}\, NAME \\
\hline
ProtectionSystem[restrictedInteraction/X, protector/P] \\
\forall\, x, y : \mathrm{dom}\, syscomp;\ i : interaction;\ a, b : Interface \bullet \\
\quad y \in restrictedInteraction\, x \Leftrightarrow \\
\quad\quad (x, a) \in \mathrm{dom}\, i.cmpIntfInstance \land (y, b) \notin \mathrm{dom}\, i.cmpIntfInstance
\end{array}
$$

Since a *RestrictedInteractionProtectionSystem* is an extension of a *BasicSystem*, all operations on *BasicSystem* can be promoted to *RestrictedInteractionProtectionSystem* using the following pattern:

PromotedBasicSystemOpeation

$$
\begin{array}{l}
\Delta RestrictedInteractionProtectionSystem \\
SomeBasicSystemOperation \\
\hline
SomePredicate \\
. \\
. \\
.
\end{array}
$$

In this case, *SomePredicate* is a predicate that relates the inputs or state in *SomeBasicSystemOp* to the state in *RestrictedInteractionProtectionSystem*. For example, an operation to create interactions on the *RestrictedInteractionProtectionSystem* is defined as shown below, with *cmpName*1? and *cmpName*2? defined as inputs from *CreateInteractionBasicSystem*.

CreateInteractionRIPS

$\Delta RestrictedInteractionProtectionSystem$
$CreateInteractionBasicSystem$

$cmpName1? \notin restrictedInteraction\ cmpName2?$

$cmpName2? \notin restrictedInteraction\ cmpName1?$

Protection through memory isolation

Another common protection mechanism is that provided by memory isolation, where each component that exists on the system is assigned a block of memory which other components must not be able to access (unless explicitly permitted). To capture this, we first specify *MemorySystem* as a component on the system that has a set of addresses, *address*, and a function, *content*, which maps each address to the data stored in it.

The contents of a *MemorySystem* can be changed by providing an address in memory and data to be stored in that address, through the *ChangeMemoryState* operation. The contents can also be read using the *ReadMemoryState* operation.

ChangeMemoryState

$OperateOnMemorySystem$
$newcontent? : DATA$

$content' = content \oplus \{memAddress? \mapsto newcontent?\}$

MemorySystem

$address : \mathbb{P}\ ADDRESS$
$content : ADDRESS \rightarrow DATA$

$\mathrm{dom}\ content \subseteq address$

ReadMemoryState

$OperateOnMemorySystem$
$memContent! : DATA$

$memContent! = content\ memAddress?$
.
.
.

OperateOnMemorySystem

$\Delta MemorySystem$
$memAddress? : ADDRESS$

$memAddress? \in address$
.
.
.

We define protection through memory isolation to include a protection function, *memoryProtected*, that specifies that one component, say x, is memory protected from another, say y, if the addresses assigned to x are not accessible to y. Here *assigned* and *access* are defined as functions mapping a component to a set of addresses, which can be changed by granting or revoking access, or assigning addresses to a particular component. Any address that is assigned to a particular component automatically

gets added to the set of addresses to which the component has access. This, however, requires that the new address has not already been assigned to another component.

MemoryProtectionSystem

$$
\begin{array}{l}
BasicSystem \\
memory : MemorySystem \\
memAccessCtr : NAME \\
access, assigned : NAME \pfun \mathbb{P}\, ADDRESS \\
memoryProtected : NAME \pfun \mathbb{P}\, NAME \\
\hline
ProtectionSystem[memoryProtected/X, memAccessCtr/P] \\
\mathrm{dom}\, access \subseteq \mathrm{dom}\, syscomp \\
\forall\, addressSet : \mathrm{ran}\, access \bullet addressSet \subseteq memory.address \\
\forall\, addressSet : \mathrm{ran}\, assigned \bullet addressSet \subseteq memory.address \\
\forall\, x, z : \mathrm{dom}\, syscomp;\ p : \mathrm{ran}\, memoryProtected \bullet \\
\quad assigned\ x \subseteq access\ x \land assigned\ z \cap assigned\ x = \emptyset
\end{array}
$$

Operations on *MemoryProtectionSystem* are promoted from both *BasicSystem* and *MemorySystem* using the promotion schemas *PromoteBSOperationMPS* and *PromoteMemoryOperationToMPS*, respectively. Operations promoted from *MemorySystem* do not change the state of the system components, connectors or interactions, whereas those promoted from *BasicSystem* cannot change aspects related to memory, such as the *access* function.

PromoteMemoryOperationToMPS

$$
\begin{array}{l}
\Delta MemoryProtectionSystem \\
\Delta MemorySystem \\
requestor? : NAME \\
memAddress? : ADDRESS \\
\hline
requestor? \in activeComponent \\
\theta MemorySystem = memory \\
memAddress? \in access\ requestor? \\
. \\
. \\
.
\end{array}
$$

PromoteBSOperationMPS

$$
\begin{array}{l}
\Delta MemoryProtectionSystem \\
\hline
memAccessCtr' = memAccessCtr \\
access' = access \\
assigned' = assigned \\
memoryProtected' = memoryProtected \\
memory' = memory
\end{array}
$$

We define a number of operations, including *GrantAccessMPS*, *RevokeAccess* and *AssignAddress*, that determine whether or not a particular component can access a given set of memory addresses.

Protection through ring architecture

The third type of protection mechanism that we consider is that provided by the ring architecture of the CPU [59]. Each component is assigned a ring level, which determines the privileges that the component will have. To capture this protection mechanism, we define *RingArchitectureProtectionSystem* to include a function *privilege* that maps a component to the *Ring* and a protection function, *ringProtected*. The protection function is such that a component, x, is protected from another component, y, if x runs in a ring with a higher privilege than that of the ring in which y runs. The ring assigned to a particular component can be changed through the *AssignPrivilege* operation, which results in a change in the set of mappings to privileges assigned to components, represented as the *privilege* function.

RingArchitectureProtectionSystem

$$
\begin{array}{l}
BasicSystem \\
cpuSubsystem : NAME \\
privilege : NAME \pfun Ring \\
ringProtected : NAME \pfun \mathbb{P}\, NAME \\
\hline
ProtectionSystem[ringProtected/X, cpuSubsystem/P] \\
\dom privilege \subseteq \dom syscomp \\
\forall\, x : \dom syscomp \bullet \\
\quad ringProtected\, x = \{y : \dom syscomp \mid privilege\, x < privilege\, y\}
\end{array}
$$

AssignPrivilege

$$
\begin{array}{l}
\Delta RingArchitectureProtectionSystem \\
assignToCMP? : NAME;\ newRingLevel? : Ring \\
\hline
sysState = live \\
privilege' = privilege \oplus \{assignToCMP? \mapsto newRingLevel?\} \\
interaction' = interaction \\
. \\
. \\
.
\end{array}
$$

Combining the Protection Mechanisms

We have so far specified a basic system as a composition of components and connectors, which can be enhanced with various protection mechanisms to control the kinds

of interactions and state changes that may occur on it. The protection mechanisms have so far been defined as individual entities. However, any non-trivial system will use a combination of these protection mechanisms, and more. For this reason, we define *ProtectedBasicSystem* as a conjunction of the protection mechanisms defined above. Below, we discuss some of the consequences of this combination.

$$\begin{aligned} ProtectedBasicSystem \mathrel{\widehat=} & \; RingArchitectureProtectionSystem \land \\ & MemoryProtectionSystem \land \\ & RestrictedInteractionProtectionSystem \end{aligned}$$

Operations, such as *StartSystemPBS*, *AssignPrivilegePBS* and *CreateInteractionPBS*, on the *ProtectedBasicSystem* are promoted from constituent systems.

Shared Resources

In the *RestrictedInteractionProtectionSystem* schema, we defined protection in terms of the non-existence of common interactions among any two components. However, in most cases components will share resources at any given time. For example, in the password manager scenario, *passman* and *passcracker* could be specified in such a way that they do not share any interaction. However, both of them might rely on a library such as *openssl* for some functionality. As a result any vulnerability in the library could be exploited by *passcracker*. It is therefore essential that the implication of sharing resources is made explicit.

$ASMP_1$: *Shared resources* — any resources that may be shared by components on the system is assumed to be free from vulnerabilities.

SharedResourceSystem

$$ProtectedBasicSystem$$
$$VulnerabilityFree, SharedResource : \mathbb{P}\, NAME$$

$$VulnerabilityFree \subseteq \operatorname{dom} syscomp$$
$$SharedResource \subseteq VulnerabilityFree$$

Direct Memory Access (DMA)

Direct memory access is a mechanism that enables DMA-capable devices to access system memory without going through the bus. This reduces access time thus increasing performance of these devices. This idea of un-mediated access to memory, however, has an implication on security. A DMA-capable device could be compromised, made to access memory assigned to a supposedly protected component, and

thus change the behaviour of that component. In the password manager scenario *passcracker* could compromise a DMA-capable device on the system and thus obtain the contents of the memory address used by *passman*. Suppose we define a *DMAEnabledProtectedBasicSystem* as a protected basic system with a set of DMA-capable devices, then we can define a DMA property as follows:

DMAEnabledProtectedBasicSystem

$$
\begin{array}{l}
ProtectedBasicSystem \\
dmaCapableDevices : \mathbb{P}\, Component \\
\hline
dmaCapableDevices \subseteq \operatorname{ran} syscomp
\end{array}
$$

$PROP_2$: *DMA* — any device included in the set of DMA capable devices has *implicit* access to all memory addresses on the system

DMAAccessibleMemory

$$
\begin{array}{l}
DMAEnabledProtectedBasicSystem \\
\hline
\forall\, dev : dmaCapableDevices \bullet memory.address \subset (access\ dev)
\end{array}
$$

Ring 0 Memory Access

The CPU ring architecture was designed to provide isolation among components of different types. It was envisioned that manufacturers would use all the four defined ring levels. However, platform manufacturers ended up using only two rings, i.e. 0 for highly privileged code such as operating system and 3 for all user space applications [59]. A consequence of this is that most devices such as device drivers end up running in Ring 0. By 2008, there were over 4 million known device drivers for Windows, with over 4000 being reported every day [40]. Most of these devices require Ring 0 access for best performance. This provides a large attack surface for malicious components. As such *passcracker* could use a compromised or poorly designed driver to exploit *passman*.

$PROP_3$: *Ring 0 memory address* — any component that is running in a higher privilege has access to all the memory used by a component running in a lower privilege. Consequently, any component running in *Ring 0* has access to all of memory.

Ring0MemoryAccess

$$
\begin{array}{l}
ProtectedBasicSystem \\
\hline
\forall\, x, y : \operatorname{dom} syscomp \bullet \\
\qquad (privilege\ x < privilege\ y \lor privilege\ x = 0) \Rightarrow \\
\qquad\quad (assigned\ y \subseteq access\ x)
\end{array}
$$

4.2.3 Trust Enablers

In this section, we identify and characterise services that enable various operations related to the enforcement of the system's behavioural properties or provision of evidence for its trustworthiness. We formalise these components in order to make explicit the properties they contribute to a system architecture and the assumptions upon which their use relies.

Integrity Measurement

In Section 2.3.7, we discussed that the process of integrity measurement provides a means for securely identifying components on a system and their configurations. To capture this, we define *IntegrityMeasurableSystem* as an extension of a *Protected-BasicSystem*, which includes a set of components that are in the trusted computing base (TCB) and support an operation, *MeasureComponentIntegrity*, to measure those components. This operation returns an *IntegrityValue*, which is meant to uniquely identify the configurations of a particular component. Therefore, we capture this as a schema comprising of the name of the component, its interfaces and computations that the component provides.

IntegrityValue

$$
\begin{array}{l}
identity : NAME \\
interface : \mathbb{P}\, Interface \\
computation : \mathbb{P}\, Computation
\end{array}
$$

IntegrityMeasurementService

$$
\begin{array}{l}
service : Component \\
recordInterface : NAME \\
\hline
recordInterface \in \operatorname{dom} service.cmpInterface
\end{array}
$$

IntegrityMeasurableSystem

$$
\begin{array}{l}
DMAEnabledProtectedBasicSystem \\
TCB : \mathbb{P}\, NAME \\
ims : NAME \pfun IntegrityMeasurementService \\
log : NAME \pfun \mathbb{P}\, IntegrityValue \\
\hline
TCB \subseteq \operatorname{dom} syscomp \\
\forall\, c : \operatorname{dom} ims \bullet c \in TCB \land (ims\ c).service \in \operatorname{ran} syscomp \\
\operatorname{dom} log \subseteq \operatorname{dom} ims
\end{array}
$$

RecordComponentIntegrityToIMS

$cmpToMeasure? : NAME$
$imsComponent? : NAME$
$\Xi IntegrityMeasurementService$
$\Delta IntegrityMeasurableSystem$

$sysState = live$

$cmpToMeasure? \in \mathrm{dom}\, syscomp$

$recordInterface \in service.activeCMPInterface$

$imsComponent? \in \mathrm{dom}\, ims$

$\theta IntegrityMeasurementService = ims\ imsComponent?$

$(syscomp\ cmpToMeasure?).measureInterface \in$
$\quad (syscomp\ cmpToMeasure?).activeCMPInterface$

$log' = log \oplus \{imsComponent? \mapsto (log\ imsComponent? \cup \{i : IntegrityValue \mid$
$\quad i.identity = cmpToMeasure? \land$
$\quad i.interface = \mathrm{ran}((syscomp\ cmpToMeasure?).cmpInterface) \land$
$\quad i.computation = \mathrm{ran}((syscomp\ cmpToMeasure?).cmpComputation)\})\}$

Integrity measurement introduces a number of properties to a system architecture and operates on a number of assumptions. We define the most important properties and assumptions below.

$ASMP_2$: *Measurement completeness* — in order to establish trust in a system, one has to determine the system in question has performed integrity measurements properly — that all components upon which the behaviour of the system depends have been measured.

MeasurementComplete

$IntegrityMeasurableSystem$

$\forall c : TCB \bullet (\exists RecordComponentIntegrityToIMS \bullet cmpToMeasure? = c)$

$PROP_4$: *Measure before execute [59]* — defines a property that if there exists an interaction in which a component was executed, then that interaction must occur after an interaction in which a component was measured.

MeasuredBeforeExecute

IntegrityMeasurableSystem

$$
\begin{array}{l}
\forall\, c : \mathrm{dom}\, syscomp \bullet \\
(\exists\, e, m : Interaction;\ RecordComponentIntegrityToIMS \bullet \\
\quad e \in interaction \land m \in interaction \land \\
\quad m.occursAt < e.occursAt \land cmpToMeasure? = c \land \\
\quad (\exists\, p : \mathrm{dom}(m.cmpIntfInstance) \bullet \\
\quad\quad (c, ((syscomp\ c).cmpInterface)\,(syscomp\ c).measureInterface) = p) \land \\
\quad (\exists\, p : \mathrm{dom}(e.cmpIntfInstance) \bullet \\
\quad\quad (c, ((syscomp\ c).cmpInterface)\,(syscomp\ c).execInterface) = p))
\end{array}
$$

Attestation

The model defined so far provides mechanisms for recording integrity measurements of components on the system. However, the model does not provide mechanisms for reporting these measurements to other entities. Grawrock [59] notes that systems without TCG-enabled services provide little or no mechanisms to provide information that enables relying entities to determine their trust in the system. Trustworthy system architectures address this problem through the process of attestation, discussed in Section 2.3.8, in which the system under consideration securely reports its configurations to entities wishing to rely on it.

To cater for attestation, we define an *AttestationService* (ATS) [72] as a component that runs on the system to provide attestations (i.e. information about the configuration of the system and its constituents). We define an *AttestableSystem* as an *IntegrityMeasurableSystem* that provides an interface for collection of attestations (i.e. provided by the *attestInterface* of the *AttestationService* component). For simplicity, we assume that the *AttestationService* has a *secret* data item, which it uses in creating attestations. In reality, such a secret could be equivalent to a combination of a *nonce* received from the challenger and an attestation identity key (AIK) stored in the TPM. We define an *Attestation* as a log of integrity values that have been recorded on the system together with a data item, *secret*, which can be used to determine the authenticity of a particular instance of an attestation.

AttestationService

$$
\begin{array}{l}
service : Component \\
attestInterface : Interface \\
secret : DATA \\
\hline
attestInterface \in \\
\quad (\mathrm{ran}(service.cmpInterface))
\end{array}
$$

Attestation

$$
\begin{array}{l}
secret : DATA \\
log : NAME \pfun \mathbb{P}\, IntegrityValue
\end{array}
$$

$$
\begin{array}{|l}
\textit{AttestableSystem} \\
\textit{IntegrityMeasurableSystem} \\
\textit{ats} : \textit{NAME} \pfun \textit{AttestationService} \\
\hline
\forall\, x : \operatorname{ran} \textit{ats} \bullet x.\textit{service} \in \\
\qquad \operatorname{ran} \textit{syscomp} \\
\hline
\end{array}
$$

We call interactions that occur over the *attestInterface* as *AttestationInteraction*. These are challenge-response interactions which result in the invocation of an *AttestSystem* operation. The *AttestSystem* operation returns integrity values from all the integrity measurement operations that occurred on the system.

$$
\begin{array}{|l}
\textit{AttestInteraction} : \mathbb{P}\, \textit{Interaction} \\
\hline
\textit{AttestInteraction} = \{i : \textit{Interaction};\ x : \textit{Interface} \mid (\exists\, a : \textit{AttestationService} \bullet \\
\qquad a.\textit{attestInterface} = x \land (a.\theta \textit{Component}, x) \in \operatorname{dom} i.\textit{cmpIntfInstance}) \bullet i\}
\end{array}
$$

$$
\begin{array}{|l}
\textit{AttestSystem} \\
\Xi \textit{AttestableSystem} \\
\textit{atsName}? : \textit{NAME} \\
\textit{attestations}! : \textit{Attestation} \\
\hline
\textit{sysState} = \textit{live} \\
(\textit{ats}\ \textit{atsName}?).\textit{service} = \textit{syscomp}\ \textit{atsName}? \\
\textit{atsName}? \in \textit{activeComponent} \\
\textit{attestations}!.\textit{secret} = (\textit{ats}\ \textit{atsName}?).\textit{secret} \\
\textit{attestations}!.\textit{log} = \textit{log} \\
. \\
. \\
. \\
\hline
\end{array}
$$

The process of attestation introduces a number of properties to a system architecture and also relies on a number of assumptions. We formalise the most interesting ones below.

$ASMP_3$: *Attestation service isolation* — in order for attestations to be meaningful, the ATS must be capable of operating without influence from other (potentially) malicious components on the system. For this reason, a trustworthy system architecture is assumed to have the ATS isolated from any other components not in the TCB.

IsolatedAttestationService

$$
\begin{array}{l}
AttestableSystem \\
\hline
\forall\, a : \mathrm{dom}\, ats;\ c : \mathrm{dom}\, syscomp \bullet \\
\qquad (c \notin TCB) \Leftrightarrow \\
\qquad\qquad ((c \in restrictedInteraction\ a) \land \\
\qquad\qquad (c \in ringProtected\ a) \land \\
\qquad\qquad (c \in memoryProtected\ a))
\end{array}
$$

$PROP_5$: *Report completeness* — attestations will be meaningful if the attestation reports are complete. We define *ReportCompleteness* to capture this property, which specifies that every program whose integrity was measured was also reported.

ReportCompleteness

$$
\begin{array}{l}
AttestableSystem \\
\hline
\forall\, RecordComponentIntegrityToIMS \mid \\
\qquad imsComponent? \in \mathrm{dom}\, ims \bullet \\
\qquad (\exists\, AttestSystem \bullet \\
\qquad\qquad atsName? \in \mathrm{dom}\, ats \land imsComponent? \in \mathrm{dom}\, attestations!.log)
\end{array}
$$

$PROP_6$: *Report consistence* — similar to the report completeness property, trustworthy systems ensure that attestation reports are consistent — every entity reported must have been measured at some point.

ReportConsistence

$$
\begin{array}{l}
AttestableSystem \\
\hline
\forall\, AttestSystem \mid atsName? \in \mathrm{dom}\, ats \bullet \\
\quad attestations!.log \subseteq log \land \\
\qquad (\exists\, RecordComponentIntegrityToIMS \bullet \\
\qquad\qquad imsComponent? \in \mathrm{dom}\, attestations!.log)
\end{array}
$$

Secure Storage

A trustworthy system stores several sensitive data such as credentials, which if compromised could impact the overall trustworthiness of the system. For this reason, this data must be protected while in use, transportation or when not in use. Trustworthy system architectures provide secure storage mechanisms, which are based on cryptographic primitives. These mechanisms involve a chain of encrypted keys where some data is encrypted with a key which itself is encrypted with another and so on. To decrypt the data, one has to be able to satisfy the requirements placed on the usage of the key at the root of the chain.

We define a *StorageService* as a component which ensures that data is only released when access conditions have been met by the requesting component. This service keeps track of the components that have access to keys used to store the data, so that when storing the data, i.e. *StoreSecurely* operation, the key used to store the data is mapped to the component requesting storage. This enables the storage service to determine whether or not to release data when requested.

We define a *SecureStorageSystem* as an extension of a *ProtectedBasicSystem*, which includes one or more secure storage services. This system requires that the component defined as part of the attestation service be one of the system components.

StorageService

$service : Component$

$keyaccess : KEY \leftrightarrow NAME$

SecureStorageSystem

$ProtectedBasicSystem$

$storage : NAME \nrightarrow StorageService$

$\forall\, c : \operatorname{ran} storage \bullet$
$\quad c.service \in \operatorname{ran} syscomp$

The main operation defined on a *SecureStorageSystem* is *StoreSecurelySSS*, which is defined as a an operation at the system level which is gets fulfiled by one of the *SroageService* component.

StoreSecurely

$\Delta StorageService$

$dataIn?, dataOut! : DATA$

$key? : KEY;\ storingCMP? : NAME$

$service.opState = running$

$keyaccess' = keyaccess \oplus$
$\quad \{key? \mapsto storingCMP?\}$

$service' = service$

PromoteStorageOp

$\Delta StorageService$

$\Delta SecureStorageSystem$

$strName? : NAME$

$sysState = live$

$strName? \in \operatorname{dom} syscomp$

$service \in \operatorname{ran} syscomp$

.

.

.

$$StoreSecurelySSS \mathrel{\widehat=} \exists\, StorageService \bullet PromoteStorageOp \land StoreSecurely$$

$PROP_7$: *Secure secrets* — sensitive information is protected such that it can only be reviewed when the policy specified at the point of storing it is satisfied.

SecuredSecret

SecureStorageSystem

$$\forall x, y : (\mathrm{dom}\ syscomp) \bullet$$
$$y \in (restrictedInteraction\ x) \land y \in (ringProtected\ x) \land$$
$$y \in (memoryProtected\ x) \Rightarrow$$
$$(\forall StoreSecurely \bullet (storingCMP? = x) \Rightarrow y = (keyaccess\ key?))$$

Trusted Isolation

Earlier we specified a number of protection mechanisms that can be used to protect one component from other components running on the system. These protection mechanisms include a component, *protector*, that is responsible for enforcing protection between any two components. However, these mechanisms do not make it explicit how such a component is itself protected. For example the *memAccessController* component defined in the *MemoryProtectionSystem* could be compromised by malicious components so as to change the way this component evaluates the *access* or *assigned* functions.

A trustworthy system architecture has to provide mechanisms that isolate such components from other components so as to prevent them from interfering with the protector. This is achieved through the concept of *trusted isolation*, where configurations for all the components critical for the behaviour of the system are recorded and any changes to these configurations are monitored and made explicit. To capture this idea, we define a *TrustedIsolationSystem* as an *AttestableSystem* such that all protectors defined for each protection mechanism are in the TCB and have their integrity recorded.

TrustedIsolationSystem

AttestableSystem

$$\{protector, memAccessCtr, cpuSubsystem\} \subseteq TCB$$
$$(\exists RecordComponentIntegrityToIMS \bullet cmpToMeasure? = protector)$$
$$(\exists RecordComponentIntegrityToIMS \bullet cmpToMeasure? = memAccessCtr)$$
$$(\exists RecordComponentIntegrityToIMS \bullet cmpToMeasure? = cpuSubsystem)$$

Secure I/O

We discussed earlier that a computer system provides protection mechanisms such as restricted interactions, which aim to limit the influence or prevent one component

from influencing the behaviour of another. However, in any non-trivial, and useful system, components will need to communicate amongst themselves or with their surroundings (e.g. display or collect data from users). In the password manager scenario *passman* interacts with a display driver to receive requests for new passwords and to send generated passwords to the user. However, *passcracker* could impersonate *passman* and fool the user into thinking that their request was serviced by *passman*.

Grawrock [59] partitions this problem into assuring the identity of humans interacting with a computer system or components or platforms at the end of a communication channel. He refers to these as trusted *path* and *channel*, respectively. We specify *SecureIOSystem* as a type of system that provides both trusted paths and channels by ensuring that all entities involved in every interaction can be surely identified.

SecureIOSystem

$$
\begin{array}{l}
IntegrityMeasurableSystem \\
\hline
\forall\, i : interaction \bullet \\
\quad (\forall\, p : \mathrm{dom}\, i.cmpIntfInstance \bullet \\
\quad\quad (\exists\, RecordComponentIntegrityToIMS;\ Interface \bullet \\
\quad\quad\quad (cmpToMeasure?, \theta Interface) = p))
\end{array}
$$

Secure Roots of Trust

The systems we have discussed so far provide certain protection mechanisms and may be able to report their configurations to enable relying entities to determine the level of protections provided by the system. However, the relying entity will have no way of determining the authenticity of the assertions or the evidence supplied from such a system. The main problem, also noted in Section 2.3.4, is that it is hard for software to prove that it is not "lying" [25]. For this reason, trusted computing provides *roots of trust* that *must* be implicitly trusted to provide a basis upon which trust in other aspects of the system, including measurement, reporting and storage, can be built.

In trustworthy systems, the roots of trust for storage and reporting are provided by the trusted platform module (TPM) [59], while the root of trust for measurement is provided by a component of the BIOS called Core Root of Trust for Measurement (CRTM) [59]. However, the impact of these roots of trust on how relying parties reach a decision about the trustworthiness of a system has not been investigated. To achieve this, we formalise the notion of root of trust and determine the properties and assumptions placed upon the two main root of trust, TPM and CRTM.

$Root ::= Reporting \mid Measurement \mid Storage$

TPM

$SRK, AIK : KEY$
$PCR : IntegrityValue$
$extendIntf, sealIntf, unsealIntf, quoteIntf : Interface$

The main operations, including *Extend*, *Seal*, *Unseal* and *Quote*, on the *TPM* are defined as follows. The *Extend* operation is the only one that changes the state of the TPM. The *Seal* operation outputs a mapping of the sealed data to the integrity values used, while the *Unseal* operation takes these mapping and returns the original data if the PCR state matches that mapped in the inut.

Extend

ΔTPM
$measurement? : IntegrityValue$

$extendIntf.intfState = enabled$
$PCR' = hash\,(PCR, measurement?)$
.
.
.

Unseal

ΞTPM
$d! : DATA;$
$sealed? : DATA \rightarrow IntegrityValue$

$unsealIntf.intfState = enabled$
$PCR \in \mathrm{ran}\, sealed?$
$d! = sealed?^{\sim}\, PCR$

Seal

ΞTPM
$d? : DATA;$
$sealed! : DATA \times IntegrityValue$

$sealIntf.intfState = enabled$
$sealed! = (d?, PCR)$

Quote

ΞTPM
$nonce? : NONCE$
$quote! : IntegrityValue \times$
$\quad NONCE \times KEY$

$quoteIntf.intfState = enabled$
$quote! = (PCR, nonce?, AIK)$

The core root of trust for measurement (CRTM) has capabilities to perform self measurements (i.e. record its own integrity).

CRTM

$selfTest : Computation$
$measureIntf : Interface$

$$\begin{array}{l}
\textbf{SecureRootofTrustSystem} \\
\hline
\mathit{ProtectedBasicSystem} \\
\mathit{tpm} : \mathit{NAME} \rightarrowtail\!\!\!\!\mapsto \mathit{TPM} \\
\mathit{crtm} : \mathit{NAME} \rightarrowtail\!\!\!\!\mapsto \mathit{CRTM} \\
\mathit{rootOfTrust} : \mathbb{P}\,\mathit{NAME} \\
\mathit{rootOfTrustFor} : \mathit{NAME} \leftrightarrow \mathit{Root} \\
\hline
\mathit{rootOfTrustFor} = \{x : \operatorname{dom} \mathit{tpm};\ r : \{\mathit{Reporting}, \mathit{Storage}\} \bullet x \mapsto r\} \cup \\
\quad \{y : \operatorname{dom} \mathit{crtm};\ r : \{\mathit{Measurement}\} \bullet y \mapsto r\} \\
\mathit{rootOfTrust} = \operatorname{dom} \mathit{rootOfTrustFor}
\end{array}$$

$$\begin{array}{l}
\textbf{PromoteTPMOperationToSRS} \\
\hline
\Delta \mathit{TPM} \\
\mathit{tpmName?} : \mathit{NAME} \\
\Delta \mathit{SecureRootofTrustSystem} \\
\hline
\mathit{tpm}\ \mathit{tpmName?} = \theta \mathit{TPM} \\
\mathit{tpm}' = \mathit{tpm} \oplus \{\mathit{tpmName?} \mapsto \theta \mathit{TPM}'\}
\end{array}$$

$$\begin{array}{l}
\mathit{ExtendSRS} \mathrel{\widehat=} \exists\, \mathit{TPM} \bullet \\
\quad \mathit{PromoteTPMOperationToSRS} \land \\
\quad \mathit{Extend}
\end{array}$$

$$\begin{array}{l}
\textbf{RecordComponentIntegritySRS} \\
\hline
\mathit{RecordComponentIntegrityToIMS} \\
\mathit{ExtendSRS}
\end{array}$$

$$\begin{array}{l}
\mathit{QuoteSRS} \mathrel{\widehat=} \exists\, \mathit{TPM} \bullet \\
\quad \mathit{PromoteTPMOperationToSRS} \land \\
\quad \mathit{Quote}
\end{array}$$

$$\begin{array}{l}
\textbf{AttestSystemSRS} \\
\hline
\mathit{AttestSystem} \\
\mathit{QuoteSRS}
\end{array}$$

$ASMP_4$: *Root of trust implicitly trusted* — every root of trust provides a foundation upon which trust in some aspect of the system can be built. Trustworthy system architectures generally operate under the assumption that every root of trust is implicitly trusted for that particular aspect of the system [59].

$$\begin{array}{l}
\mathit{ImplicitlyTrusted} : \mathit{Root} \rightarrow \mathbb{P}\,\mathit{NAME} \\
\hline
\mathit{ImplicitlyTrusted} = (\lambda\, r : \mathit{Root} \mid \\
\quad (\exists\, \mathit{SecureRootofTrustSystem} \bullet r \in \operatorname{ran} \mathit{rootOfTrustFor}) \\
\quad \bullet \operatorname{dom}(\mathit{rootOfTrustFor} \rhd \{r\}))
\end{array}$$

$$\begin{array}{l}
\forall\, \mathit{SecureRootofTrustSystem};\ c : \mathit{NAME};\ \mathit{root} : \mathit{Root} \bullet \\
\quad (c, \mathit{root}) \in \mathit{rootOfTrustFor} \Rightarrow c \in \mathit{ImplicitlyTrusted}\ \mathit{root}
\end{array}$$

4.2.4 Trustworthy System Model

In this section, we integrate the services identified in the previous section with the architecture model described in Section C.1. Figure 4.4 depicts how the trust enablers

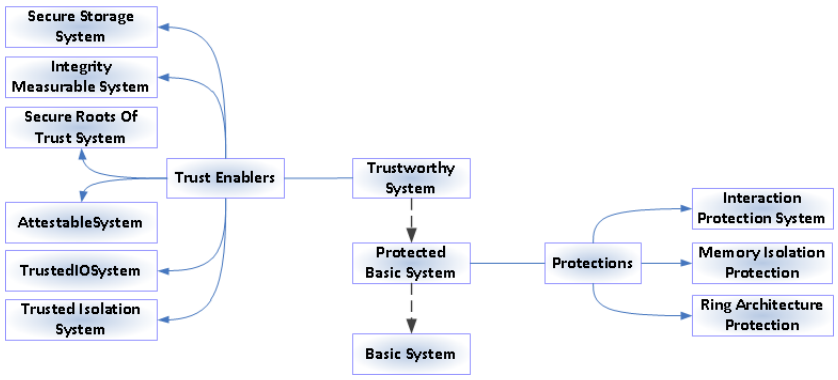


Figure 4.4: Constituents of a trustworthy system model and relationship among them

fit into the system architecture. We capture a *TrustworthySystem* as a composition of various subsystems, all of which build on the *ProtectedBasicSystem* model and provides a number of operations which can be performed by each of the subsystems. The resulting model has properties that are a combination of properties of the subsystems and assumptions upon which they operate.

A *TrustworthySystem* is a composition of various subsystems defined above, all of which build on the *ProtectedBasicSystem* model. It inherits properties from its subsystems and provides a number of operations which are fulfiled by one or a combination of its subsystems. A *TrustworthySystem* requires that all components that provide attestation, integrity measurement and secure storage are protected by one of the roots of trust.

TrustworthySystem

SecureIOSystem
AttestableSystem
SecureRootofTrustSystem
SecureStorageSystem
IntegrityMeasurableSystem

DMAAccessibleMemory

Ring0MemoryAccess

SecuredSecret

IsolatedAttestationService

ReportConsistence

ReportCompleteness

Operations on a trustworthy system are defined in such a way that each operation

can be fulfilled by one or more subsystems within a trustworthy system. To capture this idea, each operation defined above is promoted by combining it with similar operations from other subsystems. For example, to execute a component on a trustworthy system, an *ExecuteCMPTrustworthySystem* operation is defined as one that includes *ExecuteComponentSSS*, *ExecuteComponentIMS* and *ExecuteComponentATS*, implying that this operation would satisfy the preconditions of each of the subsystems included in a trustworthy system. Similarly, integrity measurement can be achieved using the *RecordCMPIntegrityTrustworthySystem* operation and attestation using the *AttestTrustworthySystem* operation. Other operations follow a similar pattern.

RecordCMPIntegrityTrustworthySystem
$\Delta TrustworthySystem$
RecordComponentIntegritySRS

ExecuteCMPTrustworthySystem
$\Delta TrustworthySystem$
ExecuteComponentSSS
ExecuteComponentIMS
ExecuteComponentATS

AttestTrustworthySystem
$\Delta TrustworthySystem$
AttestSystemSRS

4.3 Model Validation

In the previous section, we have defined various systems as compositions of architectural elements where each element is captured as a schema type, so that the state of a system is determined by the states defined for the schemas from which it is composed and its properties are determined by state invariant defined on those schemas. However, to ensure that these states and invariants are not contradictory or inconsistent, it is customary to perform type checking, prove initialisation theorems and compute preconditions of the operations.

Type checking ensures that the syntax and semantics of the Z operators used are the model is correct. As part of validation, the model was type checked using *fUZZ* [160] and some parts, which did not require rewriting the model in ISO standard, were also checked with the CZT tool [100].

In order to avoid inconsistencies in the system, it is essential to determine that the initial states are achievable. This is achieved by proving that an initial state exists for each schema [178] — the so called initialisation theorem. The model presented in this previous section is designed to be generic, and not to conform to a specific instance of a trustworthy system. Instead the model can be used to instantiate systems with

different components. This presents a challenge as far as proving initialisation theorems — prove theorems without being too specific. To overcome this challenge, we defined initialisation schemas as generic as possible and with tool support randomised the values of the other state variables not included in the initialisation. For example, to prove initialisation theorem for *Component*, we defined *InitComponent* as shown below, so that the actual number of interfaces available is left to the tool to generate.

InitComponent

$$
\begin{array}{|l}
\hline
Component \\
\hline
\forall\, i : (\text{ran}\ cmpInterface) \bullet i.intfState = disabled \\
opState = notloaded \\
cmpComputation = \emptyset \\
\hline
\end{array}
$$

Using ProZ (a sub component of ProB [131] — a tool for animating and verifying specifications written in B), schemas were initialised with random state values and animated. For example, Figure 4.5 shows that a component when initialised starts in the *notloaded* state with all its interfaces disabled. Running *LoadComponent*, changes the state of the component to *loaded*, allowing the component to be enabled for execution or measurement. The component can also be unloaded.

The use of ProZ also enabled us to evaluate preconditions for each operation. ProZ requires each specification include one initialisation schema and all visible operations should not be included in other operation schemas. In our analysis, we defined each main schema with a Initialisation schema with a unique name and created one initial schema which included a particular initialisation schema for the component we intend to analyse. For example, a TrustworthySystem is defined with an initialisation schema *InitTrustworthySystem*, which is then included in the *Init* schema as shown below.

InitTrustworthySystem

$$
\begin{array}{|l}
\hline
TrustworthySystem \\
\hline
InitIMS \land InitATS \land InitSSS \\
interaction = \emptyset \\
data = \emptyset \\
\hline
\end{array}
$$

Init

$$
\begin{array}{|l}
\hline
InitTrustworthySystem \\
\hline
\end{array}
$$

Notice how other constituent state variables such as components, connectors and active components are left to the tool to randomise and how a *TrustworthySystem*

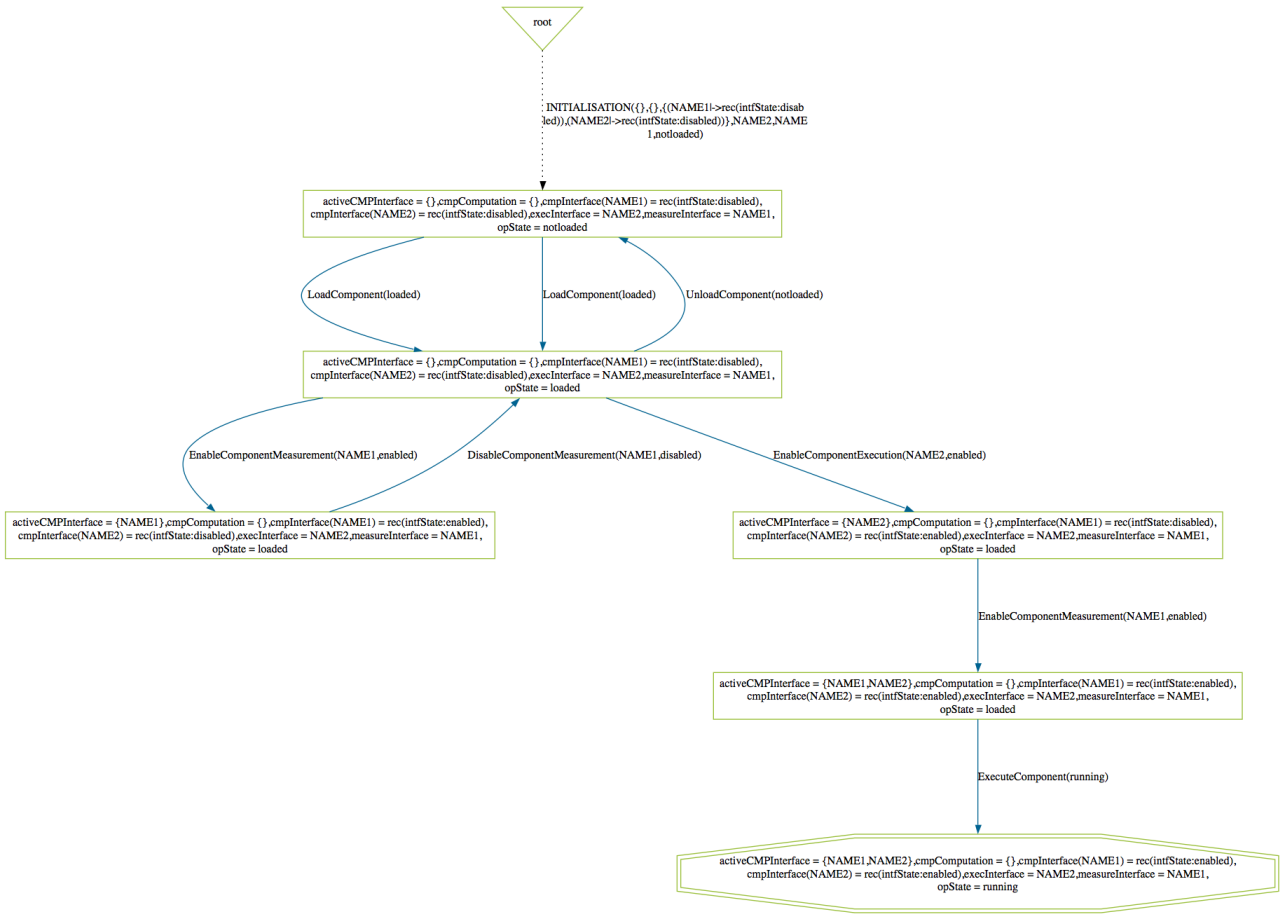


Figure 4.5: Operations that can be performed on the component and resulting state changes.

is initialisation through initialisation of its constituent systems (e.g. *InitIMS* — initialise the *IntegrityMeasurableSystem*). Once the schemas were initialised, we analysed the operations enabled at each stage and the state changes resulting from each execution of an operation. Figure 4.6 shows the history of execution of operations, operations enabled and the state of the system after a particular execution.

All schemas were animated in the same way. However, the generality of this approach quickly led to one major problem — state-space explosion. The tool was set up to find random values for each possible state variable and determine which operations would be enabled given the random values. This led to significant computations, consequently operation timeouts — sometimes the operations were performed with timeout disabled in order to get some results. We envisage that the severity of this problem would be reduced when the system is instantiated with more specific values.

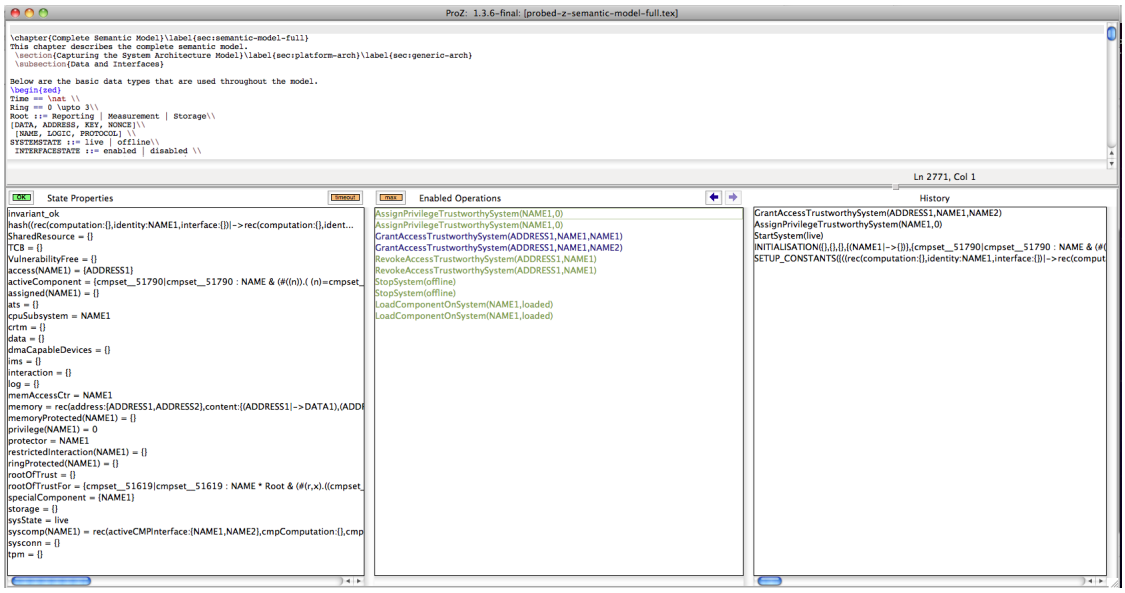


Figure 4.6: Areas of ProZ showing the execution history, state and enabled operations

4.4 Using the Model to Reason about Properties

In this section we describe a simple framework for using the model to instantiate systems, specifying trust-properties and reasoning about these properties. We apply the framework to the password manager scenario.

4.4.1 Model Usage Framework

To use the model, one needs to create appropriate bindings of the specified schemas, relate them and use the resulting model (that is with the instances linked using various relationships defined in the model) to reason about properties of interest. This is achieved by following the steps below.

1. *identify architectural elements* — this step involves identifying architectural elements that should exist in the system under consideration. This step requires one to understand the purpose of the system and decide which elements would best support that purpose. If the system under consideration has been already constructed, then this step would involve mining the elements from the design documents or system implementation. For example, a system designed to control access to some resource may include a component such as a policy enforcement point, *PEP* as part of the architecture.

2. *instantiate element* — in this step, each identified element is instantiated using an appropriate schema type. For example, suppose the *PEP* component is expected to have one interface, *PEPInterface1*, then these can be instantiated by specifying the interface that are included in the *cmpInterface* function and including this as part of the initialisation schema.

$$\begin{array}{|l} \textit{InitPEP} \\ \hline \textit{InitComponent} \\ \hline \textit{cmpInterface} = \{\textit{intf} \mapsto \textit{PEPINterface}\} \\ \hline \end{array}$$

3. *bind elements into a system* — the model includes a number of systems including *BasicSystem, ProtectedSystem, TrustworthySystem, IntegrityMeasurableSystem*, that represent one or more functionality that a system can perform. One has to, therefore, choose the appropriate type of system to work with. The choice depends on the kind of analysis desired. This step can be further divided into three steps:

   (a) decide which type of system is desired — an appropriate system has to be selected based on the kind of analysis being performed

   (b) instantiate system — an instance of the selected system is created using schema binding

   (c) link the elements to system constituents — the instances of elements created in step 2 are assigned to constituents of the system. This is achieved by creating a schema that includes the initialisation schema of the system that is to be instantiated.

   For example, suppose a *BasicSystem* was required, then the following snippet initialises a *BasicSystem* and links the instance, *PEP*, as its constituent.

$$\begin{array}{|l} \textit{InitBasicSystemPEP} \\ \hline \textit{InitBasicSystemPEP} \\ \hline \textit{syscomp} = \{\textit{name} \mapsto \textit{PEP}\} \\ \hline \end{array}$$

4. *use the linked model in reasoning* — the resulting model of the system, with all the constituents linked to the selected system instance, is used to reason about properties of interest. Reasoning is accomplished using analysis of predicate formulae.

4.4.2 Modelling the Password Manager

In this section, we revisit the password manager scenario and investigate how the model developed in earlier sections can be used to: i) demonstrate the properties that are induced by the use of a trustworthy system architecture; and ii) make the assumptions upon which the system relies explicit (i.e. why M_2 might be a better choice than M_1 in terms of its ability to prevent *passcracker* from compromising the passwords). We instantiate each of the platforms, M_1 and M_2, using the model and determine the properties and assumptions that distinguish them.

Identifying Architectural Elements

Based on the scenario description, we identify a number of architectural elements that are essential for one or both platforms. These are instantiated with the appropriate types as shown below (we ignore lower level components such as BIOS).

$$
\begin{array}{|l}
ps, malapp, drv, mem, ima, ds : Name \\
passman, passcracker, DisplayDriver, Display : Component \\
ATSServiceCMP, IMAServiceCMP : Component \\
MemorySystem : Memory \\
a1, a2, a3, a4 : ADDRESS \\
tmpName, atsName, imaName : NAME \\
IMA : IntegrityMeasurementService \\
ATS : AttestationService \\
TPMDev : TPM \\
\hline
MemorySystem.address = \{a1, a2, a3, a4\} \\
ATS.service = ATSServiceCMP \\
IMA.service = IMAServiceCMP
\end{array}
$$

System Definition

In this scenario, the analysis is focused on understanding the level of protection provided by the two platforms. Furthermore, even though M_1 is not based on a trustworthy system architecture, it will have some level of protections. For these reasons, we instantiate M_1 as a *ProtectedBasicSystem* and M_2 as a *TrustworthySystem*.

The display driver is assigned *Ring 0* while both *passman* and *passcracker* are assigned *Ring 3*. *passcracker* is assigned memory address *a3* while *passman* is assigned *a1*. M_2 builds on M_1 by specifying that its underlying protected basic system, $\theta ProtectedBasicSystem$ is equivalent to $M_!$. This means that all the bindings of elements in M_1 also apply on M_2.

*InitM*1

InitProtectedBasicSystem

$syscomp = \{ps \mapsto passman, malapp \mapsto passcracker,$
$drv \mapsto DisplayDriver, ds \mapsto Display\}$

$assigned = \{passman \mapsto \{a1\}, passcracker \mapsto \{a3\},$
$DisplayDriver \mapsto \{a4\}\}$

$access = \{passman \mapsto \{a2\}\}$

$privilege = \{passman \mapsto 3, passcracker \mapsto 3, DisplayDriver \mapsto 0\}$

*InitM*2

InitTrustworthySystem

$syscomp = \{ps \mapsto passman, malapp \mapsto passcracker,$
$drv \mapsto DisplayDriver, ds \mapsto Display,$
$atsName \mapsto ATSServiceCMP, imaName \mapsto IMAServiceCMP\}$

$assigned = \{passman \mapsto \{a1\}, passcracker \mapsto \{a3\},$
$DisplayDriver \mapsto \{a4\}\}$

$access = \{passman \mapsto \{a2\}\}$

$privilege = \{passman \mapsto 3, passcracker \mapsto 3, DisplayDriver \mapsto 0\}$

$tpm = \{tpmNAME \mapsto TPMDev\}$

$ats = \{atsName \mapsto ATS\}$

$ims = \{imaName \mapsto IMA\}$

$TCB = \{drv, ps\}$

Reasoning with the Model

There are several ways in which *passcracker* could compromise the passwords. We consider some of these below.

Driver as a shared resource

The display driver could be used by both *passman* and *passcracker* when interacting with the display system. Therefore, the display driver could be considered a shared resource. By $ASMP_1$, the display driver is expected not to have any vulnerabilities so that data exchanged with one application is prevented from reaching another application.

Password leakage through compromised driver
We have initialised the display driver to have Ring 0 privileges. Now suppose the driver had some vulnerabilities that could be exploited by *passcracker*, then by $PROP_0$, the display driver will have implicit access to both $a1$ and $a2$. It follows that if *passcracker* creates an interaction with display driver at a later point, the sensitive information accessed from $a1$ or $a2$ by the display driver could be leaked to *passcracker*.

D *Making the properties of the driver explicit*
The use of M_2 helps to identify the state of the display driver and thus any deviations from expected states can be detected. By $ASMP_2$, we note that both *DisplayDriver* and *passman* would have had their integrity measurements recorded and $PROP_4$ ensures that such measurements were performed before the component being measured had an opportunity to change its configurations (i.e. before it goes into the running state). It follows that if the driver was compromised the user would be able to detect the compromise by comparing the integrity values with known good ones.

Identifying end-points to avoid falsification of responses
When a user makes a request to have a new password generated, this request is sent to the password generator subsystem and once the password has been generate the response is sent from the password provider to the driver. M_1 does not provide mechanisms for authenticating the end points in an interaction so that *passcracker* could perform a man-in-the-middle type of attack where it pretends to be *passman*. In this case, *passcracker* would intercept the request for a new password, generate a new password and send it to the user, who would use the password without knowing that it was generated by *passcracker*. M_2 is built from a *SecureIOSystem*, which ensures that for every interaction, all end points must have been measured.

4.5 Evaluation

In this section, we evaluate the proposed approach to determine how well it meets the requirements and its benefits and challenges.

4.5.1 Revisiting the Requirements

In this section we analyse the model to determine how it addresses the requirements identified in Section 3.7.1.

`REQ 1` specifies the need to characterise building blocks of trustworthy systems. This has been achieved by following a building block approach in which elements of a generic architecture are identified and complemented with trust enablers. The characterisation has been based on a formal approach enabling precision in both the structure of the system and its properties. In this regard, our approach is a formalisation of the building blocks of trustworthy systems akin to Grawrock's [59] informal characterisation.

These properties have been demonstrated to originate from the constituents of the system, thus satisfying `REQ 2`. `REQ 3` is satisfied by the use of semantics of Z. More specifically, the use of schemas enable the elements to be specified in a domain-independent manner and thus enabling re-use. And finally, predicates can be formulated to capture trust-properties of interest and used to reason about instances of the model, thus satisfying `REQ 4`. However, a lack of adequate tool support hinders the extent to which such reasoning can be automated.

4.5.2 Lack of Support for Concurrency

The model assumes that the state changes on the system or on its constituents that occur at any particular point in time are a result of a single computation. In other words, only a single computation is performed at any given point. This is adequate for exploring the contribution of each component to the properties of a system architecture but does not address the issue of state changes resulting from concurrent execution of components. This issue is addressed in the next chapter, where a more interaction-focused model is developed.

4.5.3 Making the Basis of Trust Explicit

In Section 2.3.3 we discussed that a systematic approach for capturing the elements in a trustworthy system architecture could help to make the basis of trust in such systems explicit. In this chapter, we have proposed a semantic model that captures the elements that may exist in a trustworthy system architecture and how these elements can be combined to induce certain properties. In this model, each element that could be included in a trustworthy system architecture is formally characterised to identify how operations performed on an element could affect the state of other elements on the system. The model specifies various protections and identifies properties that each (or a combination of them) contribute to the architecture. Trust-enabling concepts

are also formalised so that the properties of the instances of the model are linked to the elements used to construct the system and thus making the basis of trust explicit.

4.5.4 Specification Clarity Versus Provability

The use of Z implies that the model benefits from Z's expressive power, so that almost any kind of behaviour can be specified using the model. For example, new schemas can be specified to capture functionality not included in the semantic model. These schemas can then be combined with the model using schema operators. However, this expressibility implies that coming up with an algorithm that enables reasoning about every possible predicate is a significant challenge [81]. This is because proofs for large specifications such as the one presented in this chapter, would require more formality and detailed checks in order to provide necessary assurances of correctness [20]. As a result one has to rely on intuition or trade between ability to perform proofs against clarity of the specifications [158]. In this dissertation we have preferred clarity of the specifications over proofs because the proofs would require significant effort but would provide relatively few benefits [103]. We imagine research with a more formal focus might concentrate on carrying out these proofs.

4.5.5 Comparison with Related Work

In this chapter we developed the model using semantics of Z. Z is mathematical language and toolset that enables precise specification of systems. Within the field of software architecture several authors have proposed models based on Z. Most notable is the work by Abowd *et al.* [3] on formalising architectural styles. The authors use Z to describe a model that forms the foundations for a framework for defining and analysing architectural styles. The first part of the model is similar to ours. It includes components connectors and configurations. The later part focuses on capturing templates for specific architectural styles such as pipe and filter. Our model and framework is different from this approach in many ways. First, our focus is one trustworthy system architectures rather than styles. As a result the semantics and level of detail considered is significantly different. We demonstrate the properties that each concept contributes to a system architecture and constraints on their combination.

Within the area of trusted computing research, a number of similar models have been proposed. More specifically, the idea of formalising trustworthy systems has been investigated by Abadi and Webber [1] and Datta *et. al.* [35]. Abadi and Webber [1] use a logic of authentication to give an account of the properties of a so-called Next

Generation Secure Computing Base (NGSCB). This approach focuses on creating statements that principals (users, software or hardware) can make or support in a system. Datta *et. al.* [35] propose a logic of secure systems (LS2) that can be used to describe low level operations in order to capture properties such as code execution. Our approach is significantly different from these approaches in that we concentrate on system-wide properties and how these are induced as a result of using certain elements in a system architecture. This is at a high-level of abstraction compared to LS2 and is not about what statements principals or elements can make, rather what the properties of these elements are.

4.6 Summary

State-of-the-art in trustworthy system architectures concentrate on using a combination of several techniques to enable trustworthy operation. However, little is known about the properties and assumptions upon which trust in such systems is based. Towards this end, in this chapter we have proposed a *semantic model* of trustworthy systems, which provides a systematic characterisation of the building blocks of trustworthy system architectures. The characterisation is based on a formalisation of the generic system architecture which is then complemented with *trust-enabling* concepts including secure storage, attestation, integrity measurement and secure roots of trust. This formalisation is akin to Grawrock's [59] informal characterisation of elements of trustworthy systems. The model defines constituents of a system, state changes resulting from operations on these constituents and properties induced from these operations or combination of elements. The model also identifies assumptions upon which these state changes or protections against them are based and thus making properties and assumptions upon which trust is based in such systems explicit.

Chapter 5

Abstraction Framework for Reasoning about Interactions

In the previous chapter, we demonstrated how a semantic model of trustworthy systems can be used to reason about properties induced by the system's building blocks. This approach enables reasoning about how the structure of a system influences its ability to operate in a specified or expected manner and is, therefore, useful for competence-based and composition-based trust-properties. However, the model is inadequate for representing interactions or runtime aspects of a system. This is because the model is not well suited for capturing concurrent executions.

In Section 3.6 we identified a mapping between interaction aspects of a system and outcome-based properties and in Section 3.7.2 we identified requirements for a model that could be used to capture such aspects. In this chapter, we build a framework that is inspired by this mapping and designed to meet the requirements discussed in Section 3.7.2. In this framework interactions that occur on a system depend on the *components* — defined as a set of processes — and their *configurations* — interfaces available and operational parameters.

This framework uses *Communicating Sequential Processes* (CSP) [144, 68] to analyse interactions resulting from parallel execution of components on a system. In addition, the use of a *refinement-based* approach enables properties to be expressed using the same semantics as the system being checked and also enables comparison of systems to some extent (i.e. any two systems can be refinement-checked with one system serving as a specification and the other as an implementation).

The framework is built on a number of design principles, which are discussed in the next section. Section 5.2 describes the abstract model, specifying its constituents and how they fit together. An approach for using the model to create instances — specific compositions resulting from initialisation of the model with its parameters —

and verifying resulting instances against outcome-based trust-properties is presented in Section 5.3. This approach is demonstrated on the trusted grid scenario of Section 3.1.2. The chapter concludes in Section 5.6 finding that suitable abstractions focusing on interactions can be used to reason about outcome-based properties.

5.1 Framework Design Principles

The architectural model described in Section 2.1.2 indicates that a system will be composed of components that interact using connectors. Each component is a composition of one or more sub-components which, similarly, interact to provide the functionality offered by the component. This means that at any given time, a system will have multiple components executing concurrently. For example, in the trusted grid scenario described in Section 3.1.2, grid middleware such as Condor [166] executes on the platform concurrently with other applications such as anti-malware or a firewall. Therefore, in order to capture interactions, we adopt an approach, based on CSP [144, 68], that inherently supports parallelism.

The framework is composed of: i) an abstract model of trustworthy systems; ii) a process for using the model to instantiate systems with varying properties; and iii) a process for specifying trust-properties and verifying them against generated instances. Before discussing the details of the framework, we begin with a description of the CSP syntax and semantics used in the model and discuss the main design principles of the framework.

5.1.1 Brief Overview of Key CSP Semantics

Communicating Sequential Processes (CSP) was developed by Hoare [68] as a notation for describing and reasoning about communicating concurrent systems. It comprises a language and mathematical models for describing systems and reasoning about properties that arise as a result of the way in which the compositions of systems interact [144]. It enables various analyses to be performed including: *determinism* — whether a system will always behave exactly the same way when given the same inputs; *liveness* — whether a system interacts with its environment; and *deadlock-freedom* — whether a system can always make progress. Most interestingly, CSP includes operators that enable subsystems to be combined into more complex systems.

CSP provides *events*, such as *write*, as primitives for atomic actions performed by a system. Events are then combined using operators into a *process* to indicate the steps a system can perform. This can be as simple as STOP — which performs no event

— or could be a combination of choices of events. The general structure of a process is an event followed (written '$\rightarrow$') by either another process or event. For example a process '`write → P`' is a process that is willing to perform the event *write* and then acts like the process P. Events may also be used to communicate data, in which case they are represented by a *channel* and the data carried on that channel. For example $write.a.d$ might be used to represent an event in which some data, d, is written to some address, a.

Operators

Operators are used to build complex processes. The *external* choice operator, `□`, enables the process to offer a choice of events to the environment while the *internal* or *nondeterministic* choice operator, `⊓`, specifies that the system selects the events to perform without influence from its environment. Other operators of interest to our model include: *hiding*, `\`, where `P \ A` is a process with all the events in the set, A, hidden; *alphabetised parallel*, `[A‖B]`, where `P [A‖B]Q` is a process such that P and Q have to agree on the events in $A \cap B$ while they are free to communicate any other event; *interleaving*, `|||`, - where `P ||| Q` is a process in which P and Q operate in parallel without agreeing on any events; and *generalised or interface parallel*, `[| C|]`, where `P [| C|] Q` is a process where P and Q have to agree on all the events in the set C.

The indexed deterministic choice, `□ x: X • P(x)`, is used in most of our process definitions. It defines a process that is willing to perform any event in the set X before acting like the process P. For example, we write '`□ code : Code • extend.code`' to represent a process that will offer a choice of *extending* any *code* from the set *Code*. A full set of operators and their properties can be found in Roscoe's book [144].

Semantic Models and Tool Support

CSP provides semantic models to enable reasoning about the properties of a concurrent system. A system's properties can be verified by checking refinement using one of the semantic models, *traces* ($\mathcal{M}_{UT}$), *failures* ($\mathcal{M}_{UF}$) or *failures-divergences* ($\mathcal{M}_{UFD}$). For example, to check that a system, S, only performs a certain sequence of events, a process that performs the desired sequence is modelled as the specification, $SPEC$, and the system, S, will serve as an implementation. Then, assertions are written using CSP and verified using the model checker FDR, a defacto standard for verifying CSP specifications, licensed by Formal Systems (Europe) Limited[1]. For

[1]Formal Systems (Europe) Limited - http://www.fsel.com

example, using the *traces* model, '`assert SPEC` $\sqsubseteq_T$ `P`' states that the process P will only perform events that *SPEC* can perform. FDR will indicate success or failure of the assertions giving counterexamples for failed assertions.

5.1.2 Principle 1: Components as CSP Processes

In the framework, components in a system are represented as CSP processes while connectors are analogous to *channels* through which processes communicate. Each process represents the kinds of messages a component is willing to communicate over specific channels and the order in which it is willing to do so (though the structure of the process may not always imply a specific order e.g. the process may be non-deterministic). A process, therefore, encapsulates the behaviour of the component on the system. For example, in the trusted grid scenario, the grid middleware could be captured as a process that is willing to accept a job, execute it on a platform, and send the results back to the requester. Similarly, the job's executable code can be represented as a process which when executed performs some function (tasks designed to achieve a certain goal) and passes the results to the executor (i.e. grid middleware).

5.1.3 Principle 2: Interactions as Synchronised Events

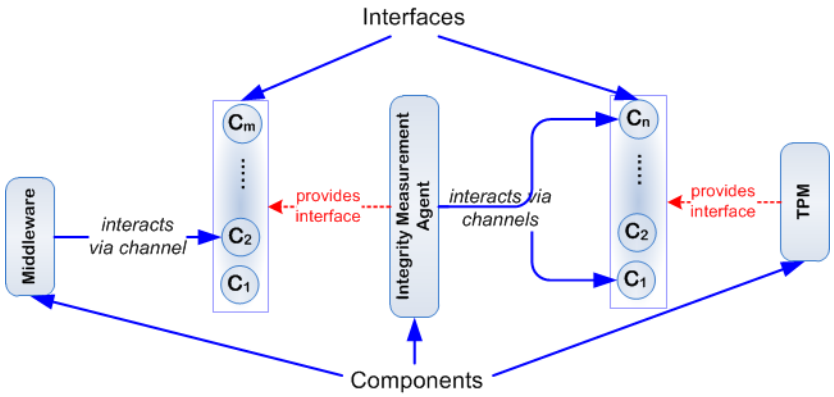


Figure 5.1: Interactions are created by overlaying interfaces among the components in the system

As discussed in the architectural model (see Section 2.1.2), an interaction is created as a result of some components engaging in communication using certain interfaces. In the framework, an interface is defined as a set of channels through which particular components can communicate. For example, if the middleware needs to record the integrity of a job, it would use one or more channels defined in the interface provided

by an integrity measurement component which would in turn use channels in the interfaces defined for the *TPM*, as illustrated in Figure 5.1.

5.1.4 Principle 3: Trust Properties Capture Absence/Presence of Interactions

As discussed above, an interaction occurs when certain conditions are satisfied. This has an important implication towards reasoning about properties of trustworthy systems. If we can determine that the channels necessary to enable interactions of interest are not available, and assuming that only those channels can enable these interactions, then we would be able to reason about the fact that these interactions would not occur. In the grid system scenario, for example, if we can determine that the channel through which unencrypted data can be copied is not available when sensitive data is available, then we can reason that sensitive data cannot be copied. Therefore, in this framework, trust properties capture the presence or absence of certain interactions or sequences of interactions.

5.1.5 Principle 4: Semantic Models as State Exploration Tool

To check the presence or absence of interactions would, however, only be possible if an entire space of the possible interactions exist. Such a space could be obtained by listing all the combinations of channels that would be available at every stage in the operation of a system together with the components offering to communicate via those channels. This is achieved by using CSP's semantic models [144]. These models enable simple process descriptions to be expanded into a state transition system that represents the possible interactions that could occur on the system together with the possible sequences in which they can occur. The operators such as generalised parallel ($[\![ ]\!]$) and interleaving ($|||$) enable simpler processes to be composed together to create more complex processes, thereby enabling component reuse and incremental design and analysis.

5.1.6 Principle 5: Trust Properties as CSP Specifications

The ability to translate process descriptions into state transition systems can be utilised to express properties without listing all the interactions of interest (i.e. those that represent the expectations captured in a property). For example, in the password manager scenario, suppose the system was expected to enforce that all passwords

generated should be encrypted when sending them to the display system, then a specification can be constructed to focus on interactions in which the password is sent to the display system. As an example, the process below specifies that before a password is sent, there must have been a request to encrypt the password and a response, unless the password was already encrypted.

```
SPEC(enc_passwords) =
        □ p: Password •
                encrypt_req.PP.CL.p → encrypt_res.CL.PP!p' → send.PP.DS.p'
                                → SPEC(union(enc_password,{p'}))
        □ p'': enc_passwords • send.PP.DS.p'' → SPEC(enc_passwords)

{* PP denotes the password provider, CL denotes a cryptography library,
DS denotes a display system and p' denotes the result of encrypting p with some key.*}
```

5.1.7 Principle 6: Refinement as a Reasoning Mechanism

Design Principles 1 and 5, discussed above, specify that components on the system and trust properties are expressed as CSP processes. This means that both the system description and the properties can be transformed into state transition systems, as discussed in Section 5.1.5. For example, the specification above can be transformed into a state transition system as shown in Figure 5.2.

A refinement-based approach allows us to prove that a given system has all the desirable properties of another system rather than simply proving that a system has certain desirable properties.

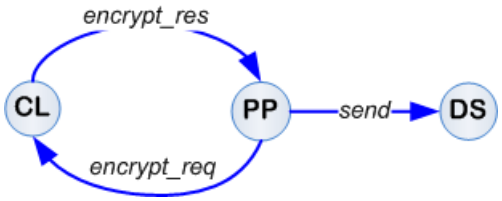


Figure 5.2: A state transition system for the SPEC process

Now, suppose we had two systems A and B, as illustrated in Figures 5.3 and 5.4, such that A sends passwords to two servers M and S after encrypting the passwords, and B sends an unencrypted password to M and an encrypted one to S. In this case, we would like to detect the fact that system B breaks our trust property, i.e. it sends an unencrypted password to another component (M). To achieve this, we can make use of refinement, so that system B would fail the specification because it performs an event (i.e. sends an unencrypted password) that is outside the sequences of events permitted by the specification. The sequences of events leading to a failure

of a particular specification is given as counterexamples, which makes it explicit why a particular system cannot satisfy a given trust property.

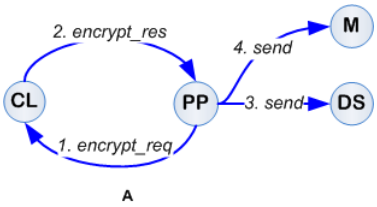


Figure 5.3: System A sends encrypted passwords to multiple components

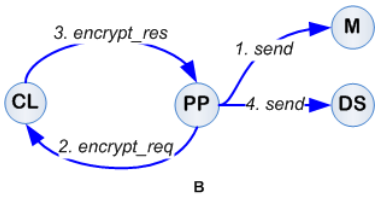


Figure 5.4: System B sends one unencrypted password before sending an encrypted password to S (step 4)

5.2 An Abstract Model

In response to the lack of fundamental elements in a platform architecture that protects against attacks [150], a trusted platform has been designed as a fundamental building block for enabling trustworthy computing [59]. A trusted platform is designed to ensure that: (i) programs running on it are isolated; (ii) processes running in user space are separated from privileged processes; (iii) data can be securely stored on it without comprising both its integrity and confidentiality even in the presence of malicious code; (iv) it can securely record and report its own configurations; and (v) immutable components (rooted in hardware) are provided as foundations for trust in the system [59]. These attributes make trusted platforms crucial in the development of trustworthy systems [30, 94, 149, 41].

Therefore, in order to reason about properties of trustworthy systems, we need to be able to understand how trust develops through the use of trusted platforms. In order to achieve this, we propose a model for capturing a trusted platform in terms of the interactions that occur on it. The model is designed to: (i) be *generic* — enabling capturing of any system that falls within the definition of a trustworthy system; (ii) be *instantiable* — allowing concrete implementations to be created from it; and (iii) enable *varying trust-levels* — different levels of trust can be assigned by the relying party based on the properties that the instance satisfies. The model is used, as described in Section 5.3, to create instances of trustworthy systems that interact with other trustworthy systems and external components in a trustworthy system architecture.

The model is based on CSP and treats a platform as a potentially malicious composition of sub-systems that interact using well defined interfaces. This enables

us to capture the idea that a system's ability to satisfy trust-properties depends on the kinds of interactions that can be created from a certain construction and configuration of its constituent elements. In the rest of this section, we discuss the constituents of the model and how they are combined.

5.2.1 Defining a Trustworthy System

In the proposed model, we define a trustworthy system as a hierarchical composition of sub-systems that interact by exposing interfaces or using those exposed by other sub-systems. Following our design principles outlined in Section 5.1, a trustworthy system is characterised as follows:

1. it is constructed from sub-systems, which are specified as CSP processes according to *Principle 1*;
2. its subsystems interact through defined interfaces, as specified in *Principle 2*; and
3. it includes one or more subsystems that provide foundations of trust in some aspect of the system, i.e. roots of trust.

5.2.2 Model Structure

The model includes subsystems, which are themselves a composition of one or more components that interact in a manner determined by the subsystem's internal structure. At a high-level, the subsystems include *Resources*, *Runtime-System*, *Applications* and *Configurations*, which interact as depicted in Figure 5.5.

The *Runtime System* interacts directly with *Resources* using the *resource_interface* and controls access to resources by providing a restricted *service_interface* through which *Applications* can interact (indirectly) with *Resources*. *System_Configurations* are provided to all the components via the *config_interface*. These are combined in the *System* process, which takes as arguments configuration information including initial memory contents, application logic and assignment of memory locations to components.

```
System(Sys_Config_Data,Res_Config_Data,Apps_Config_Data) =
        (
                (Runtime_System [|resource_interface|] Resources(Res_Config_Data))
                [|service_interface|]
                Applications(Apps_Config_Data)
        )
        [|config_interface|]
        System_Configurations(Sys_Config_Data)
```

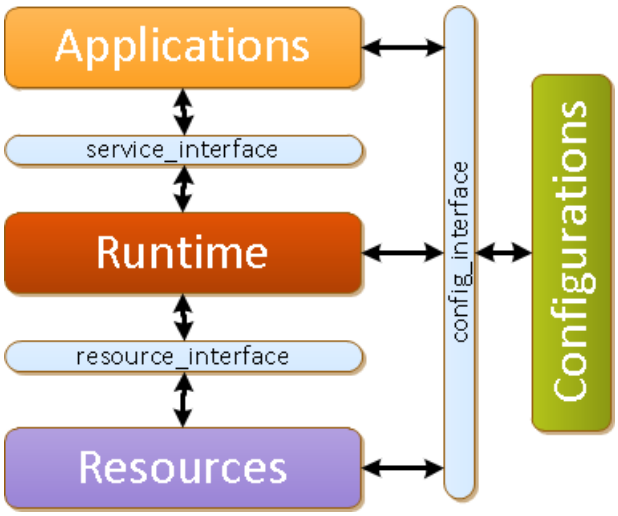


Figure 5.5: Conceptual view of the model

```
{*
Where:
Res_Config_Data is a tuple of the form (Memory Access map, Initial Memory Contents)
Apps_Config_data is a sequences of the form <(appCode, role, logic, interfaces),..>
Sys_Config_data is a sequence of the form <(Role, Memory Address),...>
*}
```

In the following sections, we discuss the structure of each of these components. We first summarise the main semantics of the elements used in the model in Table 5.1.

5.2.3 Runtime System

The *Runtime System* captures the life-cycle of a platform from start-up through the period in which applications are executed up to the point when the platform is reset or powered down. It comprises of a number of components that interact to ensure that services are available to applications when requested and that appropriate access is granted to resources. These include the *BIOS*, *BootLoader* and *Kernel* at the lowest level and an application manager and a number of services that provide trust functions such as integrity measurement and reporting and secure storage, as illustrated in Figure 5.6.

Launch Process

When a system is initialised, the *Runtime System* starts as a sequential process and later branches to support parallel execution of applications. It loads the *BIOS* which performs some *selftests* before loading the boot loader machine code, *btlcode*, from some memory location *btlloc*. The loaded *btlcode* is measured, through the

Element	Description	Example
Channels	Events occur over channels, where each channel is denoted with the actual data, role of the component performing it or the actual code.	`load.bcode.btl.btlcode`: specifies that some code, *bcode* loads a component with the role *btl* to run code *btlcode*
Choice of code	To reduce on state space, we create subsets of various data types to reduce the number of choices of data available for use in an event.	`□ btlcode: PotentialBtlCode •` specifies a subset containing code that can act as a boot loader, its size could be as small as two, one for the malicious code and the other for the non-malicious code
Identity	Represents the actual binary code of components on the system. This is distinguished from roles so that multiple components may have the same role, but each identity is required to be unique.	`read.M.knlloc.knlcode` specifies that the identity of the kernel that has been read is *knlcode*
Roles	A number of roles are defined to keep track of the expected responsibility of a particular component. This is mainly to support dynamic configuration of the model.	`config.bcode.btl.loc` : indicates that the configuration being obtained is for a component with the role of boot
Passing control	The run channel is used to receive control of execution. This channel is expected to be the first to be executed when control is passed to a component.	`Kernel(id) = run.knl.id → ..`

Table 5.1: Semantics of the elements of the model

extend.btlcode event before control is passed to the *BTL* process. The *BTL* process reads the kernel image, *knlcode*, from memory location, *knlloc*, loads and measures it before passing control to the *Kernel* process.

The *Kernel* process *loads*, *measures* and *runs* an application manager, *apmcode*, followed by an integrity measurement service, *imacode*, and then allows the application manager to run concurrently with other trust services, discussed below. The application manager executes as a recursive process that reads application code from memory, loads and runs it. When a particular application is run, control is passed to the process specified as its application logic.

```
Runtime_System  = □ bcode: PotentialBiosCode • init → selftest.bios.bcode →
        (□ loc : PotentialBtlLoc • config.bcode.btl.loc →
        (□ btlcode: PotentialBtlCode • read.M.loc.btlcode → load.bcode.btl.btlcode →
        extend.btlcode → BTL(btlcode)))

BTL(id) =  run.btl.id → ( □ knlloc : PotentialKnlLoc • config.id.os.knlloc →
        (□ knlcode: PotentialKnlCode • read.M.knlloc.knlcode → load.id.os.knlcode →
        extend.knlcode → Kernel(knlcode)))

Kernel(id) = run.knl.id → (□ apmloc: PotentialAPMLoc •
        ....
        load.id.app_man.apmcode → extend.apmcode → run.app_man.apmcode →
        ....
                        load.code.ima.imacode → extend.imacode → (run.ima.imacode →
```

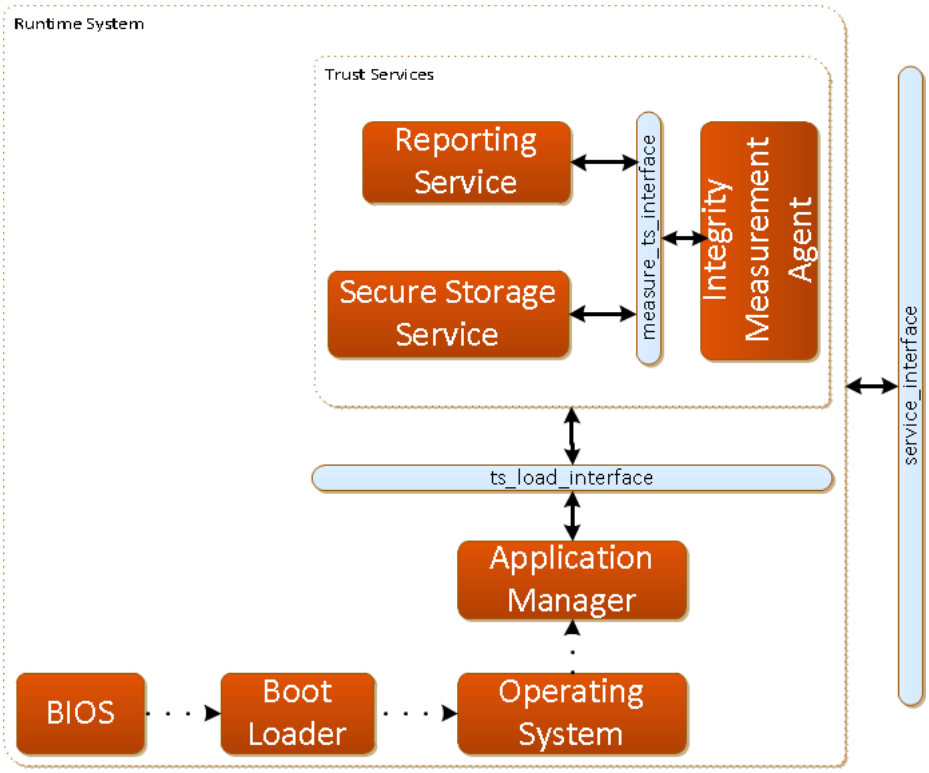


Figure 5.6: Structure of the runtime system

```
            (AppManager'(code,id,{imacode})[|ts_load_interface|] Trust_Services))
        ))))

AppManager'(id,osid) = □ loc: PotentialCodeLoc • □ rol: PotentialCodeRole •
        ....
        read.M.loc.code → load.id.rol.code → run.rol.code → AppManager'(id,osid)
```

Trust Services

A number of services, referred to as *Trust Services*, are included as part of the Runtime System to provide trust-related functions. The services are based on trusted computing functionality, including integrity measurement, reporting of the system state and secure storage of data. This is captured by the *Trust_Services* process, which is a combination of the processes *Integrity_Measurement*, *Secure_Storage* and *Reporting_Service*.

The *Integrity_Measurement* process extends the loaded code before it is run. This is achieved by synchronising on the events, *load* and *run* performed by the application manager when executing applications, or the kernel when loading other trust services. This process uses the check if $member(rol, TcbRoles)$ to ensure that only components defined as part of the TCB are measured.

The *Secure_Storage* process provides secure storage by *sealing* the data to some state and allows retrieval of the stored data. This is achieved through the *seal* and *unseal* channels, respectively. The *Reporting_Service* process accepts and responds to integrity challenges. These services interact with the rest of the runtime system using the *ts_load_interface*.

```
Trust_Services = (Service(s3,Secure_Storage) ||| Service(rs,Reporting_Service))
        [|{|run.role.x | role ← {s3,rs}, x ← PotentialTrustService|}|]
        Service(ima,Integrity_Measurement)

Service(role,service_logic) = □ code : PotentialTrustService •
                              run.role.code → service_logic(code)

Integrity_Measurement(id) = □ rol: PotentialCodeRole •
        ...
          load.ld.rol.c → (if member(rol,TcbRoles)
                then extend.c → run.rol.code → Integrity_Measurement(id)
          else run.rol.code → Integrity_Measurement(id)))

Secure_Storage(id) = □ ap: diff(IMA_Measurable,{Garbage}) •
        □ data: Secret • □ blb : Blob •
        (store.id.ap.data → seal.data.ap.blb → stored.id.ap.blb →
        Secure_Storage(id)
        □ retrieve.id.ap.blb → unseal.blb.data → retrieved.id.ap.data →
        Secure_Storage(id))

Reporting_Service(id) = □ n': Nonce • challenge.n' →
        (□ q : Quote • quote.n'.q → response.q → Reporting_Service(id))
```

Runtime Configurations

The model is designed such that the *AppManager* process is unaware of the order in which applications will run and instead relies on the *System_Configurations* process, outlined in Section 5.2.6, to enforce the order. The *AppManager* simply loads the configuration and executes an application according to the configuration settings obtained for that particular application. This implies that an application can be specified separately from its configuration enabling different instances of a system to use the same model but different configurations.

5.2.4 Resources

Resources provide mechanisms for components to interact, share data, or perform computations. For example, an application that stores a secret will use the storage service, provided as part of trust services, which in turn uses the TPM, which is a resource, to store the secret. Similarly, an application that needs to communicate to another component has a choice of sending the message over a network or use via shared memory.

Several types of resources may exist in a system. However, to keep the model simple, only two resources, a *TPM* and a memory system, are considered.

Trusted Platform Module

We define a *TPM* as a resource that offers four basic operations of *extend*, *seal*, *unseal*, and *quote*. Though in reality these operations are complex, involving a number of intermediate steps such as loading keys, the model abstracts away from such details and concentrates on the core functionality of integrity measurement, secure storage and secure report. The *TPM* is assumed to start in a "*Good*" state which may change as operations, more specifically *extend*, are performed. This is in fact an assumption made in real systems (i.e. when the system boots up, the PCRs are reset to some initial values). The actual functionality is achieved through the functions *getQuote*, *createblob* and *getUnsealedData*.

```
TPM = □ c : Code • extend.c → TPM_Op(newhash(Good,c))

TPM_Op(s) = (□ c : Code • extend.c → TPM_Op(hash(s,c))) □
        (□ n': Nonce • quote.n'.getQuote(n',s) → TPM_Op(s))□
        (□ d: Secret • seal.d.c!createblob(d,c) → TPM_Op(s))□
        (□ b: Blob • unseal.b!getUnsealedData(b) → TPM_Op(s))
```

Memory System

Memory is an important part of any computing device. In our model, memory is modelled as a process, *Memory System*, that provides three main features: reading, writing and configuring access rights. These are achieved through the channels, *read_req*, *write_req* and *config_req*, respectively. Each of these operations will only succeed if the requesting application has access rights to perform the operation on the requested address. These permissions can be changed using the *config_req*.

Responses are sent over the corresponding response channels, *read_res*, *write_res* and *config_res*. These channels return *Ok* or *Reject* depending on whether the requested operation was permitted. In the case of a read response, if the operation was unsuccessful, *Garbage* is returned instead of the actual data contained in the location requested.

```
.....
 □ A : diff(Component,{M}) • □ d : Data • □ l : Location • □ r : Right •
    write_req.A.M.(d, l) →
    (if (canWrite(A,l,WriteAddresses)) then
          Memory_System_0'(M,mem_Map,fn_update(content,l,d),A,Ok)
       else
          Memory_System_0'(M, mem_Map,content,A,Reject)
    )
   □
    read_req.A.M.l →
```

```
    (if (canRead(A,l,ReadAddresses)) then
            Memory_System_0''(M,mem_Map,content,fetch(content,l),A,Ok)
     else
            Memory_System_0''(M,mem_Map,content,Garbage,A,Reject)
    )
□
config_req.A.M.(l,p) →
    (if (canConfig(A,ConfigAccess)) then
            Memory_System_0'''(M,updateAccess(mem_Map,(l,p),A),content,A,Ok)
     else
            Memory_System_0'''(M,mem_Map,content,A,Reject)
    )
            ......
```

Combining Resources

We consider resources to be passive elements which only respond to requests from other elements of the system. This means that resources do not interact on their own. For this reason, we combine the resources using interleaving so that resources do not have to synchronise on any events. We define the Resources process as one that takes configurations specifying the contents of memory and access control matrix.

```
Resources(cfg) = TPM ||| Memory_System(cfg)
```

5.2.5 Applications

The components of the model discussed thus far do not enable the system to provide any useful functionality to users. To support this, we specify applications as a composition of processes that run in parallel to and use interfaces provided by the runtime system.

Single Application

Each application is specified as a tuple in the form (identity, Application Logic, Application Interfaces), where Application Logic is a process representing the events that the application is designed to perform. The application logic synchronises on interfaces provided by the runtime system as well as those provided by other applications.

Composing Applications

The model supports dynamic specification of applications that run on the system, so that the composition of applications is specified as part of the model while the actual application logic is passed to the model as part of the configurations performed during the instantiation process. Each application logic may use any desired channels as part

of the process description, with a restriction that channels in the *service_interface* are strictly for use between application-to-service interactions.

Algorithm 1: DynamicApplicationsProcess

input : Sequence, S, of tuples of the form (*identity*, *application logic*, *app interface*)
output : A CSP process combining all application logic using specified interfaces

```
1  if S has elements then
2      A_1 ← head(S)          //get the first application;
3      if #S == 1 then
4                             //S is a singleton sequence;
5          P ← A_1.2(A_1.1) return P;
6                             //return the only application process in the sequence;
7      else
8          A_2 ← head(tail(S))  //get the second application;
9          Σ_A1 ← A_1.3         //get the application interface of A_1;
10         Σ_A2 ← A_2.3         //get the application interface of A_2;
11         P ← A_1.2(A_1.1) [Σ_A1 || Σ_A2] A_2.2(A_2.1) //synchronise A_1.2 and A_2.2 on Σ_A1 and Σ_A2;
12         S ← tail(tail(S))    //get the remaining applications;
13         while S has elements do
14             A_3 ← head(S)          //get the next application;
15             Σ_A3 ← A_3.3           //get the application interface of A_3;
16             Σ_P ← Σ_A1 ∪ Σ_A2      //get the application interface of P;
17             P ← P [Σ_P || Σ_A3] A_3.2(A_3.1) //synchronise P with next application;
18             S ← tail(S)            //remove one application from sequence;
19 else
20     P ← SKIP ;
21 return P;
```

Algorithm 1 provides a means of composing applications with the rest of the system. The algorithm accepts a sequence of application specifications and returns a process representing synchronised applications. Each element of the sequence is a tuple consisting of application identity, application logic and a set of channels over which the application can interact with other applications on the system. It iterates through the sequence and creates a process by synchronising each application logic to the consecutive application logic using their respective interfaces. The resulting *Applications* process is placed in parallel with the *Runtime_System*, as discussed in Section 5.2.2, and is of the form:

```
Applications(<(Id_A, A,I_A), (Id_B, B,I_B), (Id_C, C,I_C),..> = ((A(Id_A) [I_A||I_B] B(Id_B) ) [I_A∪
I_B||I_C] C(Id_C))..
{*Where A, B, and C are application logic and I_i are interfaces for each application*}
```

5.2.6 System Configurations

We define configurations as data used to: i) initialise components within the system; ii) control the order in which components execute; and iii) control the interfaces through which components interact. This is captured using the *System_Configurations* process, which provides an interface through which configuration information can be obtained for various components within the system. This process is created by se-

quentially composing all the configurations of individual subsystems. Each part of the configurations is specified as a process that takes a sequence of tuples specifying the *role* of a component and the *location* in which its code is stored. This process provides this information to other components using the *config* channel.

```
System_Configurations(ComponentLocs) =
        Configuration(getComponentLocation(ComponentLocs,btl));
        Configuration(getComponentLocation(ComponentLocs,os));
        Configuration(getComponentLocation(ComponentLocs,services));
        Configuration(getComponentLocation(ComponentLocs,applications))

Configuration(ConfigurationSeq) = □ code: PotentialCfgLoader •
        if null(ConfigurationSeq) then SKIP
        else    let (role,loc) = head(ConfigurationSeq)
                within
                config.code.role.loc → Configuration(tail(ConfigurationSeq))
```

5.3 Framework Usage

In this section we discuss how the framework makes use of the model of the previous section to create instances and verify their properties.

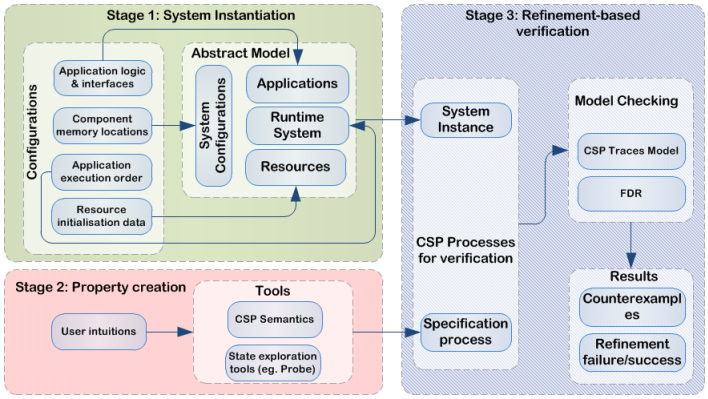


Figure 5.7: Stages in the use of the abstraction framework

A three-stage approach is adopted as illustrated in Figure 5.7. The first stage involves generating system instances from the abstract model. This is achieved by passing initialisation data including the logic of applications that will run on the system, the order in which they run, their locations in memory and the contents of memory. This results in a process representing the events that the system is willing to accept from its environment. This process can then be combined with other systems or external agents such as users within the environment.

In the second stage, the user generates processes that will act as specifications in the third stage. These processes represent the order of events necessary to capture a

particular trust-property and are generated using semantics of CSP. More specifically, a process using data type definitions and CSP operators is created based on the user's understanding of what a particular property means in terms of the events permitted.

The specification process and system instance are then verified using a refinement-based approach. This step involves performing trace-refinement with the process capturing the property acting as the specification and the system instance as the implementation. These are passed to model checking tools, more specifically *FDR*, which give results indicating whether or not refinement passes or fails and gives counterexamples in the case of failure.

5.3.1 System Instantiation

Having described the abstract model, we may ask the question, *how do we create more concrete implementations from it?* The main idea is to provide the actual values of all the configuration settings and the actual logic of the applications that run on the system. Specifically, one has to provide: i) initial memory content; (ii) locations of the components to be loaded on the system; and iii) application logic for each application. These are passed as parameters to the *System* process, i.e. *System(Memory_Configuration, Res_Init_Data, Apps_Logic_and_Interfaces).*

Component Locations and Order of Execution

Memory configurations specify a mapping, (*RoleId*, *Address*), between components on the system and the locations in which these are expected to be found. Rather than specifying where each executable code is located, these configurations specify where each component *role* is expected to be found. For example, rather than specifying that the *linux 2.6 kernel* will be found in location X, they specify that code for a kernel should be loaded from X without specifying which particular kernel will be loaded. The actual kernel that is loaded will be determined by the contents found in the specified memory location. This helps to capture the idea that component locations may contain malicious code.

Memory configurations are also used to specify the order in which the components can be loaded. This is enforced by providing the configurations in a sequence, *Configuration_Map*. The *System_Configuration* process discussed in Section 5.2.6 makes use of this sequence to determine which configurations, i.e. location information, are available to other components through the *config* channel. A *Configuration_Map* $=< A, B, C, .. >$ implies that configurations A will be available

on the *config* channel before configurations B and, therefore, a component relying on B will have to wait to execute until configurations A have been passed. The complete specification of memory configurations is in the form:

$$Configuration_Map = \{< c_1, c_2, ..., c_n > | \forall i : [1, n] \bullet c_i \in \mathbb{P}(Role, Address)\}$$

Resource Initialisation

Resource initialisation involves specifying the initial contents of memory on the system. This is achieved by specifying a map of locations to the data contained in each location as shown in *Initial_Mem_Map*. Currently, the model is built to support one memory device and thus automatically assigns the content to that device. It can easily be extended to support multiple memory devices, for example by specifying a map of maps such as *Mem_Component_Map*.

$$Initial_Mem_Map = \{(l, d) \mid l \in Address, d \in Code\}$$
$$Mem_Component_Map = \{(id, I) \mid id \in Identity, I \in \mathbb{P}\, Initial_Mem_Map\}$$

Specifying Applications

The *System* process expects a parameter *Apps_Logic_and_Interfaces* to be passed as part of the initialisation process. This parameter specifies the applications that will exist on the system together with interfaces through which each application can interact with other applications. The parameter conforms to the input of **Algorithm 1**, i.e. a sequence of tuples, where each tuple is in the form (*App_Id*, *App_Logic*, *App_Interface*). Therefore, to specify applications, one provides the identity of each application together with a process representing the actual events that the application is expected to perform and a set of channels that the application should synchronise on with other applications.

Application logic is specified using the generic structure *App_Logic_Process* (shown below). Each application logic is a process that represents the actual execution (e.g. protocol) that the application performs and observes the restriction on the use of the *service_interface*. For example, an application, *app*1, that simply prints "hello" when it runs would be specified as `App_Logic(app1) = print.app1.hello → SKIP`, where *print* is a channel that connects to some display unit.

$$App_Logic(id) = App_Logic_Process \mid id \in Code, App_logic_Process \in ValidCSPProcess$$

5.3.2 Property Specifications

The framework is intended to enable verification of outcome-based properties. To support this, properties are specified using the same semantics used for the abstract model. That is, each property is expressed as a CSP process with the desired behaviour. To come up with such a process, the user makes use of CSP semantics, including its data types and operators, and their intuition about how to express the desired or undesired outcomes in terms of events that can be performed by the system under investigation. For example *SPEC_tcb* — every component whose role is in the TCB is measured — and *SPEC_mb4r* - – every component that is measured is measured before it is executed — can be expressed as processes as follows.

```
SPEC_tcb(TCBRoles) = □ c: Code • rol: Role • load.id.rol.c →
        if member(rol,TCBRoles)then extend.c → SPEC_tcb(TCBRoles)
        else SPEC_tcb(TCBRoles)
SPEC_mb4r = □ c: Code • rol: Role • id : Identity •
        load.id.rol.c → extend.c → run.rol.c → SPEC_mb4r
```

5.3.3 Verification Model

A refinement-based approach for verifying properties is adopted. Using this approach, we model the system instance, obtained from the process described in Section 5.3.1, as the implementation and the process capturing the property, as described above, as the specification. For example, the abstractions for the above specifications for a platform, P, and the refinement checks would be performed as follows:

```
--for SPEC_tcb
system1 = P\diff(Events,{|load,extend|})
assert SPEC_tcb ⊑_T System1
--for SPEC_mb4r
system2 = P\diff(Events,{|load,extend,run|})
assert SPEC_mb4r ⊑_T System2
```

The implementation may require some abstraction so that only events necessary to verify the property of interest are exposed. This requires considerable care in choosing which events to hide because a property may fail or pass simply because some crucial events are not exposed or unnecessary events have been exposed.

Refinement checks are performed using CSP's semantic models and the model checker FDR. The results obtained are mainly in two forms: i) failure or success; and ii) counterexamples. A failure is obtained when the system instance is found to be capable of performing events that are not permitted by the specification. In the case that the specification is written to capture the existence of an adverse outcome, then failure implies that the system will *not* produce the adverse outcome, whereas success implies that the system will perform such an outcome. Counterexamples are

also produced in the case of failure. These are used to provide details about what led the system instance to fail a particular check. Again, if the specification checks for the presence of an adverse outcome, counterexamples will show how the system prevents such outcome.

5.4 Application on a Trusted Grid Architecture

We consider the application of the proposed model and framework on the trustworthy grid platform scenario of Section 3.1.2. We base our platform construction on the architecture proposed by Cooper and Martin [31]. They argue that middleware in a grid architecture may be compromised and should, therefore, not be part of the trusted computing base (TCB). They propose an architecture in which a grid platform receives an encrypted job together with a job security manager (JSM) and runs the JSM which enforces security policies on the job and attests to the execution of the job. In this section, we specify two platforms suitable for this architecture where the first runs malware while the second runs the expected code. We begin by presenting the desired properties for such a platform before discussing a set of processes that are common to both platform instances. The platform instances are then presented and their properties verified.

We define the following variables to represent code that will run on the platforms. We use *eveware_5x* to represent malicious code, *Garbage* to represent meaningless information and *Good* and *Bad* for *quotes* coming from the TPM. Other variables include the boot loader, *grub_4_2*, kernel code, *linux_2_6*, integrity measurement software, *ima1_2*, reporting service, *ats_5*, grid middleware, *gt_5*, a job security manager, *jsm_1_9*, storage service, *rserv_1_8*, application manager, *appman1_2*, and a job, *snstv_job*.

5.4.1 Property Specification

The architecture is expected to ensure that the key used for encrypting/decrypting the job can only be released when a trustworthy TCB (including the JSM) is run. This property can be specified by splitting it into parts: a *good* trace, ($PROP1$) — specifying that the correct result, $R1$, can only be obtained if no malicious code was present in the TCB — and a *bad* trace, ($PROP2$) — specifying that if malicious code is executed then *Garbage* will be received as the result, allowing the verifier to detect that something went wrong.

```
PROP1(tcb) = □ mdwid : PotentialMdw • □ jid : JobId •
     if not null(tcb) then extend.head(tcb) → PROP1(tail(tcb))
     else comm_results.jsm_1_9.mdwid.jid.encrypt(R1,K) → SKIP

PROP2(tcb) =  extend.head(tcb) → PROP2'(false,tail(tcb))
        □ extend.eveware_5x → PROP2'(true,tail(tcb))

PROP2'(isMal,tcb) = □ jsmid : PotentialJSM • □ k : JobKey •
     □ mdwid : PotentialMdw • □ jid : JobId •
     if not null(tcb) then
             if length(tcb) == 1 and not isMal
             then extend.eveware_5x → PROP2'(true,tail(tcb))
             else extend.head(TCBSeq) → PROP2'(isMal,tail(tcb))
                 □ extend.eveware_5x → PROP2'(true,tail(tcb))
     else comm_results.jsmid.mdwid.jid.encrypt(Garbage,k) → SKIP
```

5.4.2 Common Setup

In Cooper's architecture, a platform consists of middleware (*Middleware_Run* or *Mal_Middleware_Run*), an application that runs a job (*Job_App_Run*) and a job security manager (*Jsm_App_Run*). These applications interact with an external verifier or user (*Verifier*). We specify the logic for each of these applications as follows.

Middleware gets the encrypted job from the verifier and forwards it to the JSM via the *comm_job* channel. We specify a malicious middleware that attempts to run the wrong job as *Mal_Middleware_Run* in the same way as *Middleware_Run* except that it replaces the contents of memory location, $l8$, which is expected to contain the job security manager.

The JSM then obtains the key from the verifier via the *key_request* channel, sending along a *quote*. A decryption key is obtained from the verifier via the *key_response* such that if the quote indicates that the system is in a bad state, then *Garbage* is sent as the key, instead of the actual decrypting key. The JSM passes the decrypted job binary code to the job execution application by synchronising on the *execute_job* channel. The results obtained from the job are sent to the middleware and then to the verifier via the *comm_results* channel.

```
Middleware_Run(id) = □ enc_j: EncryptedJob • □ jid: JobId •
        □ U: ExternalAgent • get_job.U.id.jid.enc_j →
        (□ jsmid : PotentialJSM • comm_job.id.jsmid.jid.enc_j →
        (□ enc_r: EncryptedResult • comm_results.jsmid.id.jid.enc_r → SKIP))

Mal_Middleware_Run(id) = □ enc_j: EncryptedJob • □ jid: JobId •
        □ U: ExternalAgent •  get_job.U.id.jid.enc_j → write.M.l8.eveware_5x →
....

Jsm_App_Run(id) = □ mdwid: PotentialMdw • □ enc_j: EncryptedJob •
        ...
        (□ q: Quote • response.q → key_request.id.ks.jid.q →
        (□ k: JobKey • key_response.ks.id.jid.k →
        (□ r: Result • execute_job.jid.decrypt(enc_j,k).r →
        comm_results.id.mdwid.jid.encrypt(r,k) → SKIP))))
```

```
Job_App_Run(id) = □ j: Job • □ r: Result • □ jid: JobId •
                execute_job.jid.j.getResult(j) → SKIP

Verifier(id) = □ mdwid: PotentialMdw • □ enc_j: EncryptedJob •
        ....
        □ q: Quote • □ enc_r: EncryptedResult •
        key_request.jsmid.id.jid.q →
        if q == Good then key_response.id.jsmid.jid.K →
                comm_results.jsmid.mdwid.jid.enc_r → SKIP)
        else key_response.id.jsmid.jid.Garbage →
                comm_results.jsmid.mdwid.jid.enc_r → SKIP)
```

The memory configurations are also specified, as shown below, to indicate the locations and order of execution of the components on the system. We also specify a sequence *TCBElements* to indicate the components expected to be in the TCB, i.e. these components determine whether or not the correct key is released from the verifier.

```
Memory_Contents = {(l1, grub_4_2),(l2, linux_2_6),(l3, appman1_2),
        (l4, ima1_2), (l5, ats_5), (l6, rserv_1_8), (l7, gt_5),
        (l8, jsm_1_9),(l9, snstv_job),(l10, eveware_5x)}

TCBElements = <grub_4_2,linux_2_6,appman1_2,ima1_2,ats_5,
                rserv_1_8,jsm_1_9,snstv_job>

SPEC_good = PROP1(TCBElements)
SPEC_bad = PROP2(TCBElements)
```

5.4.3 Instantiating a Malicious System

A malicious platform is configured to run a malicious grid middleware, which replaces the *JSM* with a malicious one that does not enforce the policies as intended by the job owner. This allows the job data to be corrupted or stolen.

```
Mal_App_Config= <(mdw,Mal_Middleware_Run,mdw_intf),(pep,Jsm_App_Run,pep_intf),
                                (app,Job_App_Run,job_intf)>
Mal_System_Cfg = System_Configurations(Mal_Systen_Loc_Cfg)
Mal_System_ResourceCfg = (M,Memory_Contents)
Mal_System_Loc_Cfg       = <(btl,l1),(os, l2),(app_man,l3),(ima,l4),
                        rs,l5),(s3,l6),(mdw, l10),(pep,l18),(app, l9)>

Mal_System = System(Mal_System_Cfg,Mal_System_ResourceCfg,
                        Mal_App_Config)
```

5.4.4 Instantiating a Non-Malicious System

A non-malicious system is configured to load the desired (not *eveware*_5x) code for each component. Each of the applications *mdw*, *pep* and *app* are executed in line with the specified application logic.

```
NonMal_App_Config = <(mdw,Middleware_Run,mdw_intf),...>
NonMalSystem_Cfg = System_Configurations(NonMal_Loc_Cfg)
NonMal_ResourceCfg = (M,Memory_Contents)
NonMal_Loc_Cfg     = <.....
```

```
                    ...,(mdw,l7),...>

Non_Mal_System = System(NonMalSystem_Cfg, NonMal_ResourceCfg,
                            NonMal_App_Config)
```

The two systems differ in that *Mal_System* has a malicious middleware application, *Mal_Mdw_Run*, and the location specified for middleware is $l10$ which contains malicious code represented by *eveware_*$5x$.

5.4.5 Property Verification, Results and Discussion

The two systems specified are verified against the properties specified in Section 5.4.1 using refinement as outlined in Section 5.3.3. Abstractions are created in the same way for both platforms using the $Abstract4Props1N2$ function, which takes a process and returns another process with events on all channels other than *extend* and *comm_result* hidden.

We use the *stable failures* model for checking $SPEC_{good}$ because not only does it show what events the platform will perform but also what it will refuse to perform, referred to as *refusals* in CSP. However, for $SPEC_{bad}$, the *trace* model suffices as it is enough to show that the process can perform untrustworthy actions.

```
Abstract4Props1N2(P)    = P\diff(Events,{|extend,comm_results|})

Abstracted_Non_Mal_Systen = Abstract4Props1N2(Non_Mal_Systen)
Abstracted_Mal_Systen     = Abstract4Props1N2(Mal_Systen)

assert SPEC_good ⊑_T Abstracted_Non_Mal_Systen   --label:GoodOnNonMalicious
assert SPEC_good ⊑_T Abstracted_Mal_Systen       --label:GoodOnMalicious

assert SPEC_bad ⊑_T Abstracted_Non_Mal_Systen    --label:BadOnNonMalicious
assert SPEC_bad ⊑_T Abstracted_Mal_Systen        --label:BadOnMalicious
```

Malicious Platform Results and Discussion

The malicious platform fails on $SPEC_{good}$ giving a counterexample trace shown below. On the other hand the assertion on $SPEC_{bad}$ passes.

$\langle extend.grub_4_2, extend.linux_2_6, extend.appman1_2, extend.ima1_2,$
$extend.ats_5, extend.rserv_1_8, extend.eveware_5x, extend.snstv_job,$
$comm_results.eveware_5x.eveware_5x.j1.Encrypt.(Garbage, Garbage)\rangle$

Listing 5.1: Result — counterexample trace for assertion labelled 'GoodOnMalicious'

The trace includes an *extend.eveware_*$5x$ indicating that the *JSM* was replaced with *eveware_*$5x$. A further analysis of this system indicates that when *eveware_*$5x$ requests for a key pretending to be *JSM*, the verifier detects that the platform is not trustworthy because the quote sent on the *response* channel is *Bad*. The verifier then sends *Garbage* as a key for decrypting the job, which when used to decrypt the job

gives *Garbage*. The result of executing *Garbage* is also *Garbage* and hence the result sent back being $Encrypt.(Garbage, Garbage)$. Therefore, the malicious platform fails to satisfy $SPEC_{good}$ because some TCB components are not trustworthy.

Non-malicious Platform Results and Discussion

The non-malicious platform fails on $SPEC_{bad}$ but passes $SPEC_{good}$. The counterexample indicates that no malicious code was executed as part of the TCB, a condition enforced by $SPEC_{bad}$. The non-malicious platform never executes untrustworthy code in the TCB and never sends *Garbage* as a result.

$\langle extend.grub_4_2, extend.linux_2_6, extend.appman1_2, extend.ima1_2,$
$extend.ats_5, extend.rserv_1_8, extend.jsm_1_9, extend.snstv_job\rangle$

Listing 5.2: Result — counterexample trace for assertion labelled 'BadOnNonMalicious'

5.5 Evaluation

The previous sections have covered the framework and its usage has been demonstrated on a trustworthy grid architecture. In this section, we evaluate the benefits and challenges of the framework.

5.5.1 Satisfying the Requirements

In Section 3.7.2, we identified a number of requirements that a model for capturing interactions aspects of the system should satisfy. In this section, we analyse the model to determine how it satisfies the requirements.

`REQ 5` stated the need to capture components involved in an interaction. The model achieves this by representing components as CSP processes that communicate among themselves. This has a further advantage in that multiple processes can be included and allowed to execute concurrently, satisfying `REQ 6`. Furthermore, the use of CSP semantics implies that the model can take advantage of inherent support for parallelism.

`REQ 7` specified the need to support systems with varying properties. This chapter achieves this by using a framework which separates the model from configuration data, so that different configurations can be passed to the model to yield systems whose properties depend on the configurations passed. `REQ8 8` is satisfied by the provision of counterexamples in the case verification fails, while `REQ 9` is satisfied by the use of a refinement-based approach for varying properties. As such, both the property and

the system instance can be expressed as processes which can be translated into state transition systems whose sequence of events can be compared.

5.5.2 Supporting Virtualised Systems

Virtualisation has emerged as one of the main enabling technologies for supporting isolation among systems. Indeed, several trustworthy systems [94, 149] make use of virtualisation to provide an environment in which one or more virtual machines or applications can be executed in such a way that all applications and virtual machines that are part of the TCB are measured.

The model assumes a non-virtualised environment but could easily be modified to support virtualised systems. This would involve modifying the *Runtime System* so that a virtual machine monitor (VMM) is loaded in place of the operating system kernel. Then virtual machines can be executed on top of the Runtime System where each VM has an identity and interacts with the kernel using some interface, such as *knl_vm_interface*. The virtual machines would then be interleaved and placed in parallel with the rest of the components of the model.

Resources, TPM and Memory System, would also need to be modified to keep track of the virtual machine instance making use of the resource. This can be achieved by passing an additional parameter to the TPM and Memory, as well as channels used by these processes to indicate the identity of the VM.

5.5.3 Managing the State Space

One of the main challenges in any modelling activity is reducing the state space [110]. Too much information in the model could lead to an explosion of state space, making it difficult to verify the properties as available tool support may fail to explore the entire state space.

In the design of the model, several strategies were adopted to reduce the state space. For example, rather than including all the possible events that can be performed by the TPM, we chose to include only those that are essential for demonstrating the properties of interest such as integrity measurement. Furthermore, rather than specifying all the possible values for a data type, we chose to restrict the set available by creating several subsets for a particular purpose, for example, *PotentialBTLCode*, rather than *BTLCode*.

5.5.4 Comparison with Related Work

CSP is designed to enable reasoning about the properties of concurrent systems that communicate [144] and is, therefore, ideal for scenarios such as the ones considered in Section 3.1. It has been successfully applied towards analysis of critical applications including security [143, 95], fault tolerance [21] and safety-critical systems [27]. This success combined with its inherent support for parallelism and communication, makes it an ideal tool for our scenario.

The work presented in this chapter comes closest to that by Rohrmair [142] and Lyle [98] who both use CSP to model some aspect of trustworthy systems. The differences and shortfalls of these approaches, compared to our proposal, were discussed in Section 2.4.3.

Other possible alternatives that were investigated including BAN logic [23] and a logic of secure systems [35] both proved inadequate. This is because their inability to support concurrent execution meant that interactions resulting from existence of multiple components on a system could not be identified.

5.6 Conclusion

Outcome-based properties are best determined when an interaction-oriented approach is followed. Such an approach requires a means of capturing the components that exist on the system and the entire set of interactions that may result from their concurrent execution. Furthermore, because systems designed in the same way could still produce different outcomes, it is important that the effects of applying different configurations on the possible outcomes are explored.

This chapter has proposed the use of an abstraction framework based on CSP to capture the construction and configuration of system and to reason about how these influence the system's properties. The framework separates the model of a system from its configuration allowing instances with varying properties to be created by applying different configurations.

The framework's ability to express both the properties and the system using the same semantics imply that it can be extended towards comparing properties of any two systems. However, when using the framework, care must be taken to ensure that appropriate abstractions are created and that suitable events are selected to prevent state space explosion.

Chapter 6

Runtime-based System Trustworthiness

In Chapter 3, we identified four classes of trust properties and mapped each class to some aspect of the system that could be used to reason about properties in that class. We have so far proposed a *semantic model*, in Chapter 4, to enable reasoning about properties that fall into the *competence-* and *composition-based* classes. In the previous chapter, we described an *abstraction model and framework* that provides a means for reasoning about *outcome-based* properties. These models are concerned with construction of the system and the interactions resulting from a particular configuration of the system. However, they are limited in their ability to capture events that have already occurred on a system and to relate these to possible future behaviour, a necessity for reasoning about properties that fall in the fourth class, *shadow of the future*.

Taking the view that an architecture is the *runtime abstraction* [44] of the system, this chapter demonstrates that information collected at various phases of a system life-cycle can be used to create runtime abstractions that enable reasoning about *shadow of the future* properties. The chapter proposes two approaches towards capturing and representing evidence and using it to reason about desired properties. The first approach is a trace-based verification model, discussed in Section 6.3. In this approach evidence collected in the form of a trace of events is transformed into a *process model*. Trust-properties are verified using a formal approach that determines whether or not the traces captured in the process model satisfy the specification — where the specification represents the desired trust-property. While this approach demonstrates that it can help reason about complex properties, we found that it requires a considerable effort and expertise to capture the desired properties.

Hence, in Section 6.4, we propose a second approach based on *provenance* that complements the trace-based model by enabling properties to be expressed as queries over an RDF graph. The proposed provenance-based verification model extends the open provenance model (OPM) [116, 115] with semantics that enable the evidence to be collected in the form of a provenance graph – a causality graph that records past executions [115] — that is suitable for verification of trust-properties. Using this graph, trust-properties are verified using a combination of the formal verification approach used in the trace-based analysis and provenance queries. Simpler properties can be verified using provenance queries while more complex properties can still be checked by transforming the graph into a model similar to the process model and verified by means of refinement.

Before presenting the proposed approaches, we begin with an analysis of the challenges involved with verifying trust properties using runtime aspects. This is followed by an identification of the relationship between trust properties and evidence in a system. We present the trace-based approach in Section 6.3 and the provenance-based approach in Section 6.4. We discuss the findings in Section 6.5 before concluding in Section 6.6.

6.1 The Challenge of Verifying Trust-properties Using Runtime Aspects

The challenge of determining how secure a system is against certain attacks and how to establish how trustworthy it is [125, 87] requires collecting reliable attestations about the system's behaviour [19].

Consider the scenario of a trustworthy collaborative system described in Section 3.1.3. Researchers wishing to use virtual appliances (VAs) developed by other collaborating researchers have the option of assuming that the VAs are in a good state — all the necessary software has been installed and configured appropriately — or to verify the state of a VA before they use it. A VA may be intentionally or accidentally configured with malicious components or insecurely leading to breach of confidentiality and integrity requirements. Additionally, it could be compromised and be used to launch denial of service attacks against other services within an infrastructure. For these reasons researchers have to go for the latter option.

Researchers need to check security configurations, integrity of the installed software and generated results, and input data and experiment models that have been used on a virtual appliance before using it. Only those satisfying the researcher's

software and security requirements should be used for further experiments. This requires researchers to collect information that can be used to reason about properties of interest, as specified in `REQ 10`. However, the nature of this information, how it relates to trust-properties, how it can be used to reason about trust-properties, and how it can be managed in distributed environments [72] has not received sufficient consideration.

This challenge has been acknowledged by the TCG and other authors [133, 153]. In an effort to support interoperability the TCG proposed integrity schemas [163], which capture information about components that could exist in a system. However, these schemas can not be used to capture the activities that may have occurred on a system or to relate them. Similarly, Presti [133] acknowledges the need to collect structured information from trustworthy systems and proposes the notion of a *tree of trust*, which Schmidt et al. [153] propose to output from a modified TPM command set, as a mechanism for representing verification data. A tree structure, however, lacks semantics to capture the events on a system in a comprehensive manner. This is because with a tree structure, any given node will have only one edge connecting to it (i.e. the connection from its parent). As a result only one relationship between a given node and its parent can be expressed thereby restricting the number of relationships that can be expressed among information collected from the system.

In Section 3.6.3, we discussed that runtime aspects of a system could provide information that can be used to determine the actions that have been performed by the system. These actions have an influence on the actions that may be performed by the system in future. For example, if a firewall is configured to block traffic on a given port, then it can be expected that any communication that relies on that port would not be permitted. In this chapter we demonstrate how a trace-based verification model and a provenance-based model complement each other to enable this information to be collected in a manner suitable for use in reasoning about trust-properties.

6.2 Trust Properties and Evidence Classification

In Section 3.7.3 we specified the need to capture information that can serve as reliable attestations about the state of the system. In this section we consider the scenario of a trustworthy collaborative system, presented in Section 3.1.3, and investigate how properties in the shadow of the future class can be further sub-divided. This helps to

identify the kinds of information, which we refer to as *evidence*, that can be collected from the system to enable reasoning about these properties.

6.2.1 Trust Properties

The *shadow of the future* class of trust properties can be further divided into sub-types as shown in Table 6.1.

Type	Description	Examples
Deployment configurations	Seek to determine whether or not a virtual appliance (VA) that will be used by a researcher will exhibit certain behavioural patterns when executed.	i) an executable will use known configurations; ii) a given executable will be executed before another executable; or iii) cryptographic keys will be reset at start-up.
Processes and parameters	Identify the processes carried out during the creation of a VA, the order in which they were performed and the parameters that were used as input to the processes.	i) a critical package was installed; ii) a given package was configured as expected; or iii) expected privileges were assigned to a given object.
Data sources and integrity	Seek to establish the authenticity of data included on a VA.	i) packages installed were downloaded from trusted sources; ii) packages installed were of a known integrity; or iii) a given file was obtained from a known package.
Process integrity	Seek to determine whether or not the software components that executed as part of the build process, described below, behaved as expected.	i) programs executed had known integrity values; or ii) a given process used a particular executable file.

Table 6.1: Sub-types of properties for the *shadow of the future* class

6.2.2 Evidence Classification

The properties identified above require evidence from different phases of a system's life-cycle, which we divide into three: *inception*, *creation* and *execution*.

Inception Phase Evidence

The inception phase involves gathering the components that make up the system architecture as well as validating and certifying their authenticity. The components are then placed in a location that can be accessed during the creation phase. For example, in the motivating case described in Section 3.1.3, components that will be used to create virtual appliances such as file system, operating system kernel and packages

are placed in a software repository ready for use during the build process. This phase roughly corresponds to the collection phase in the TCG integrity architecture [163]. Table 6.2 lists some examples of evidence that can be collected from this scenario.

	Evidence	Description
1.	Code signatures	Digital signatures that enable relying entities to determine the authenticity of code
2.	Quality evaluation	Evaluations performed to certify the quality of the system or some component, e.g. Common Criteria [78] certification
3.	Compilation process and flags	A list of switches used to control the compilation process of a piece of code

Table 6.2: Examples of evidence in the inception phase

Creation Phase Evidence

Once the components are ready, the *creation phase* can be initiated to obtain the components from their locations and install them on the target system. This phase prepares the system for execution. For example, in the collaborative system scenario, the creation phase includes setting up the build environment, installing relevant packages and performing configurations such as the database connections or network port settings. Table 6.3 lists some examples of evidence that can be collected during this phase.

	Evidence	Description
1.	build platform configuration	Configurations settings the platform used to create virtual machines
2.	components executed	A list of components that were executed on the build platform e.g. VA build tools, package managers
3.	component configurations	Settings such as ports open, digital signature checks or enabled services
4.	virtual appliance contents	The actual software packages installed or files copied to the disk image

Table 6.3: Examples of evidence that can be collected during the creation phase

Execution Phase Evidence

The final phase of the system life-cycle, *execution*, is where components installed in the previous phase are instantiated to process data and provide services for which the system has been designed. This phase might also involve installation of new components or runtime re-configuration [126] to replace failing elements, increase the

system capacity or change the configuration settings. Some examples of evidence in phase are given in Table 6.4.

	Evidence	Description
1.	input/output integrity values	Recorded integrity values for any input and output e.g. command line parameters, environment variables
2.	virtual appliance configurations	Configuration settings such as user accounts and privileges, network configurations and start-up scripts
3.	scripts that will be executed	A number of scripts can be copied to the VA file system and set-up to execute when the VA is launched
4.	software installation	A list of software installed on the system during its execution phase
5.	runtime environment configuration	Configuration settings of the runtime environment such as VMM networking options

Table 6.4: Examples of evidence that can be collected during the execution stage

The rest of this chapter develops mechanisms for representing and using this evidence to reason about properties that fall into the *shadow of the future* class.

6.3 Trace-based Verification Model

In this section we describe the first of our proposals to use the evidence identified above in reasoning about trust-properties. The proposed model, referred to as a *trace-based verification model*, is based on translating a trace of events collected from a system into a model that is then analysed to determine the existence or non-existence of patterns or sets of patterns of events. More specifically, models that are a representation of the runtime system are created by observing events that occur on the system and then verified to determine whether they exhibit any behaviour that deviates from expectations.

6.3.1 Assumptions

Before we describe the model, we first state the assumptions upon which it relies:

`ASMP 1` a trustworthy logging system capable of collecting the logs securely and protecting the log integrity is available.

`ASMP 2` the logging system mentioned in *ASMP 1* is capable of logging all of the informations described in Section 6.2.2 and proving to remote entities that the logs are integrity-protected.

`ASMP 3` an access control mechanism is in place to ensure that only authorised entities can access the evidence collected from the system.

6.3.2 Overview

The approach is based on the idea of modelling the trace of events collected from a system as a process model — a representation of activities performed on the system, the order in which they were performed and the components that performed such activities — and performing refinement checks against a pre-defined whitelist — a repository of specifications which capture trust-properties. This approach, illustrated in Figure 6.1, involves creating runtime abstractions from the event logs, generation of specifications and multiple runs of trust-property verification on the obtained abstractions.

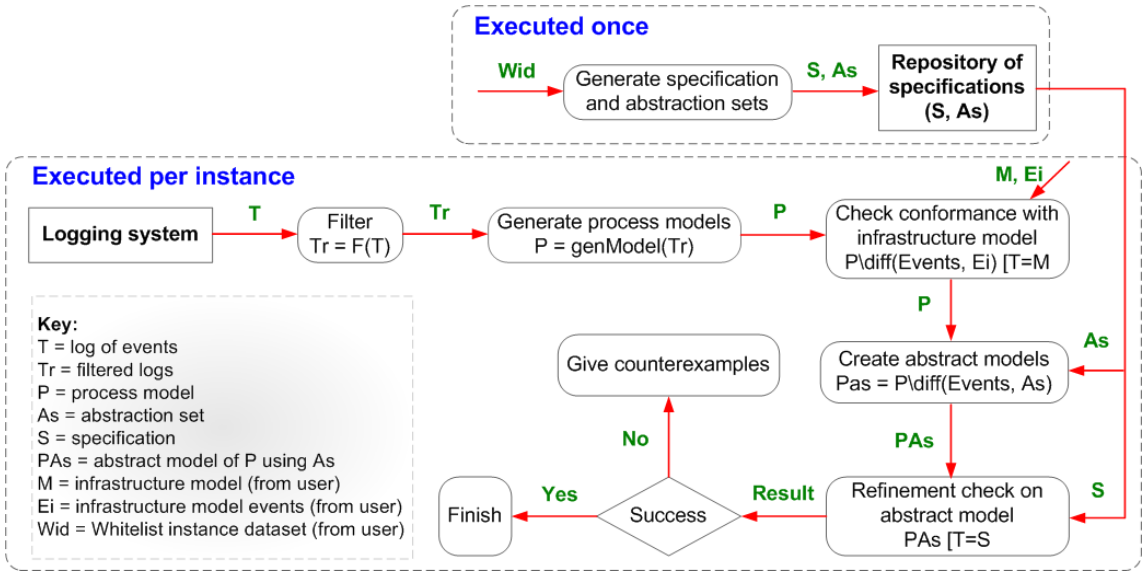


Figure 6.1: Overview of the trustworthiness verification process

The specification generation process, described in Section 6.3.3, takes a dataset that conforms to the whitelist schema, defined in Section 6.3.3, and creates processes that capture intended properties together with a set of events relevant for proving each property.

Event logs are collected from the system while it is running using the logging system mentioned in `ASMP 1`. The logs are then integrity checked before unwanted events are filtered, based on the properties of interest. The filtered logs are then transformed into a process model, described in Section 6.3.4, which is verified against the properties captured by the specifications in the repository, described in Section

6.3.5. This is achieved by creating system abstractions, using a process described in Section 6.3.5, suitable for verifying specific properties, and checking whether the system abstractions refine the specifications. When there is a refinement failure, counterexamples are given, indicating the sequence of events that led to the failure. These processes are described in more detail in the sections that follow.

6.3.3 Whitelist as a repository of specifications

The concept of a *whitelist* has been defined as a means of providing *known good configurations* against which integrity values collected from a system can be compared [72]. We extend this concept to enable the provision of references for known-good *sequences of events* as well as *acceptable combinations of configurations and events*. We thus define a whitelist *as a repository of specifications that capture expected sequences of events and/or combinations of configurations and events that occur on a system*. We propose a schema for the whitelist, that extends the TCG integrity schemas [163], and an approach for generating specifications from data conforming to this schema.

Specification of the whitelist schema

The TCG Infrastructure Working Group defines a Reference Manifest (RM) Schema [163] — as an extension of the Core Integrity Schema — for specifying reference measurements of components that may be used in integrity verification. This schema, however, can not capture *inter-relations* between components, such as the order in which one component must be executed relative to another, that exist in a given environment. We extend the RM schema to capture various forms of inter-relations that may exist between components of a system. This schema, described in Appendix A, includes a number of entities, such as *ConigurationDataType*, *DependencyTypes* and *ComponentRoleType*, which are composed to create a single entry in the whitelist, *WhitelistEntryType*.

Specification generation

The specifications in the repository are generated through a process that takes a dataset conforming to the whitelist schema, referred to as the *whitelist instance dataset*, and returns a set of specification processes and associated *abstraction sets*. It is divided into the following stages:

1. *Instance decomposition* — involves extracting parts of the whitelist instance dataset that are required for a given property, and passing them on to an appropriate property-specific specification generator. Suppose we are interested in generating a specification that captures a property "every software component installed on a machine has a known integrity". Given a whitelist instance dataset, WL_i, we first specify that unique identifiers, *UUID*s, are required for this property. These are extracted from the parent *RimmType* of every *WhitelistEntryType* and sent to the property-specific specification generator.

2. *Property-specific specification generator* — a generator takes parts of the *WhitelistEntryType* and creates processes that capture the intended property. In our example, the processor would take a set of *UUID*s, S, and return a process which enforces that every *UUID* passed as data on the *install* channel must exist in S. This would be specified as a specification in CSP as follows:

   $SPEC_{exist_on_WL}$(S) = x:S • install.x → $SPEC_{exist_on_WL}$(S)

   If the order of installation was important, then *DependencyType* and *OrderType* would also be extracted and included in the specification.

3. *Abstraction set generation* — involves creating a set of events, an *abstraction set*, that are essential for reasoning about a given property. In our example, an abstraction set, A = {|install|}, indicates that only the events that occurred on the *install* channel would be used. The abstraction sets are then combined with the related specifications as tuples $(SPEC, A)$, and stored in the repository.

6.3.4 Trace-to-process transformation

Given an observed execution trace, S_o, a process model, P_s, that is capable of producing S_o is generated. P_s is then checked for refinement against the specifications in the repository. In this section, we describe how P_s is obtained from S_o.

The transformation of an execution trace into an equivalent process model can be done in multiple ways. One way would be to use denotational semantics of CSP, where the observed trace is transformed to the most non-deterministic process that chooses to perform the events in the given trace [144]. Roscoe [144] defines such a process, P, using the *traces* model, as follows:

```
P(<>)      = STOP
P(<tick>)  = SKIP
P(<a>^s)   = a → P(s)
```

Furthermore, given any set of traces, S_o, a process, $P(S_o)$, capable of generating all the traces in the set can be defined as a non-deterministic choice of all the $P(s)$, where $s \in S_o$:

```
P(S_o) = ⊓ s:S_o • P(s)
```

For example, suppose the observed execution traces, S_o, is such that:

```
S_o = {<install.firewall, install.tomcat, config.firewall.default, reboot>,
       <start,install.java, install.scientific_app, shutdown>}
```

Then the resulting process would be:

```
P(S_o) = install.firewall → install.tomcat → config.firewall.default → SKIP
       ⊓ install.java → install.scientific_app → SKIP
```

Notice that some events in the trace such as *reboot* or *shutdown* may be translated to *SKIP*, while conditions such as *crash* may be translated to *STOP*, indicating successful operations or errors, respectively.

This approach has limited practicality because the processes are not optimised for model checking. An alternative approach would be to use process mining techniques [171] to extract process models from the execution traces. A log file obtained from the system can be transformed into a process model using a framework such as Process Mining (PROM) [172] and its associated toolkit [170]. A plug-in such as the BPMN plug-in defined within the toolkit is used to create a BPMN process, which in turn, translated into a CSP model using existing techniques proposed by Wong and Gibbons [176] or Flavio et al. [45].

6.3.5 Property Verification

We have discussed how specifications and associated abstraction sets are generated and how process models are created from event logs. The question that remains is, how these are used to reason about *shadow of the future* properties? First, specifications capture the desired properties. Then the properties are verified using refinement, where the process models act as implementations and the specifications are obtained from the whitelist. This process involves creating suitable abstractions of the process models and refinement against the whitelist.

System abstractions

Not every event that occurs on a running system is necessary to prove a given property. For example, if one wanted to check what components were installed on a system, then only installation-related events are essential. Therefore, we need to create representations of the system that would allow us to focus on the events necessary to reason about a given property. We propose the use of abstractions to generate these representations. For example, if we consider $P1$ (from the example above), then the system abstraction is a representation that hides all events other than those on the *install* channel. System abstractions are created using CSP's abstraction mechanisms with *abstraction sets* serving as the visible events. Suppose we have a process model P_S, and an abstraction set A, then the system abstraction, P_{S_A}, would be generated by hiding all the events that are not in A:

P_{S_A} = P_S \ diff(Events,A)

Refinement against the whitelist

Finally, the abstract process model is checked for refinement against the specifications in the repository. The specifications used here must be mapped to the abstraction set that was used to create the system abstraction being checked. Given a mapping of specification to abstraction set, $(SPEC, A)$, the verification is performed as follows:

SPEC [=T P_{S_A}

This can be checked using FDR by specifying it as an assertion. The results obtained are similar to the results obtained from the refinement checks discussed in Section 5.3.3.

6.3.6 Verifying Virtual Appliances in a Collaborative System

In Section 3.1.3 we described a scenario in which researchers use virtual appliances to share ready to run experiment environments. In Section 6.1 we discussed the need for researchers to verify the security and software configurations of these virtual appliances. In this section we demonstrate how the trace-based verification approach discussed above can be used to achieve this.

A private cloud system was setup using Ubuntu Cloud Server[1] as shown in Figure 6.2. This comprised of two compute nodes, each running a *Node Controller*, and one machine serving the role of a *Cloud Controller*, *Cluster Controller* and *Storage*

[1]http://www.ubuntu.com/cloud/private

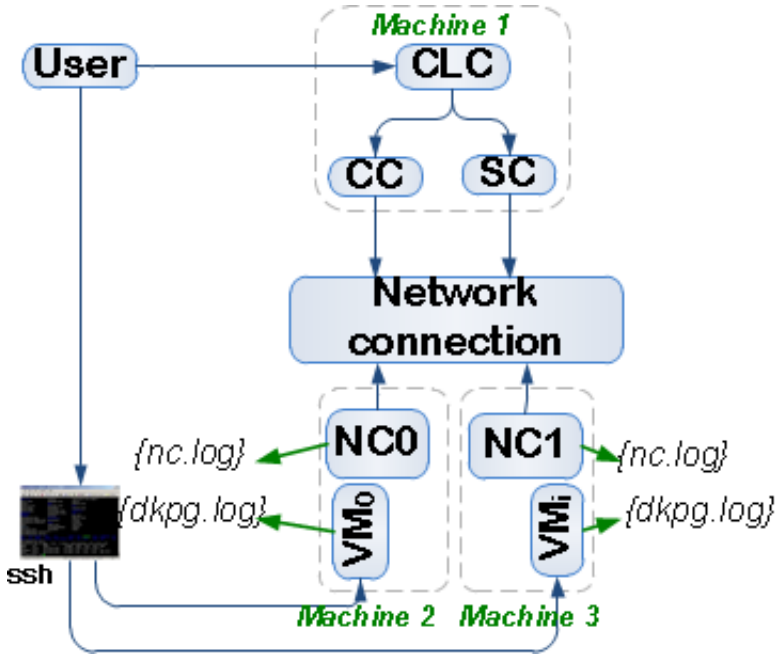


Figure 6.2: An experiment to collect and analyse event logs from virtual appliances

controller. Then two initially identical virtual machines were instantiated in the cloud setup. Software installation events performed through the package manager were collected through *dkpg.log* and virtual machine management events through *nc.log* — the node controller log file. The list of events that were performed on each virtual machine is shown in Table 6.5.

	events on VM1	events on VM2
1.	install *firewall* firestarter	install *firewall* Lokkit
2.	install *webserver* apache	config *firewall* (Custom settings)
3.	config *firewall* (Default settings)	install *webserver* apache
4.	config *webserver* (Webserver-Restricted)	config *webserver* (Webserver-Restricted)
5.	install *app* Scientific Webapp	install *app* Scientific Webapp
6.	execute *firewall*	execute *webserver*
7.	execute *webserver*	execute *firewall*
8.	execute *app*	execute *app*

Table 6.5: List of events performed on the virtual machines

CSP implementation of the generator

A CSP implementation of the specification generator was developed according to the model described in Section 6.3.3. It was designed to generate specification processes that enable us to reason about the following properties:

1. A known firewall was installed with the correct configurations and activated before running the experiment.

2. The expected application was used to run the experiment.

A whitelist instance dataset was defined to include two firewalls, *Firestarter* and *Lokkit*, with acceptable configurations ($Defaultsettings = \{HTTP = 1, SMTP = 0, Telnet = 0, SSH = 0, FTP = 1\}$) and a web server, *Apache*, with configuration ($Webserver - Restricted = \{Includes = 0, ConfigAccess = 0, FollowSymLinks = 0\}$). A simple application, which we called *Scientific webapp*, was defined to have dependencies on the installation and configuration of a firewall and the execution of a web server. This resulted in a number of specification processes, some of which are shown below and described in Table 6.6.

Specification Name	Description
SPEC_firewall_installed	Specifies that *some* firewall was installed on the system. If the installed firewall was uninstalled at some point, then another one or the same should have been installed later.
SPEC_app_config_dependence	a web server should have been had a restricted configuration, i.e. $Includes = 0, ConfigAccess = 0, FollowSymLinks = 0$, before it was executed
SPEC_firewall_configured	the firewall was configured with the default settings, i.e. $HTTP = 1, SMTP = 0, Telnet = 0, SSH = 0, FTP = 1$
SPEC_app_install_dependence	a firewall and a web server should have been installed before the *scientific app* was installed
SPEC_app_execute_dependence	specifies that a firewall was executed before the web server was executed which should have also been executed before the *scientific app* was executed

Table 6.6: A description of some of the specification processes

```
SPEC_firewall_installed =
        □ f: FIREWALL • install.firewall.f → (
        uninstall.firewall.f →
                □ y: FIREWALL • install.firewall.y →
        ....

SPEC_app_install_dependence =
        □ f: FIREWALL • install.firewall.f → (
                □ w: WEBSERVER • install.webserver.w → install.app.ScientificWebApp →
        ....

SPEC_firewall_configured =
        □ f: FIREWALL • config.firewall._.FirewallDefault →
        ....

SPEC_app_config_dependence =
        □ w: WEBSERVER • config.webserver.w.WebserverRestricted → execute.webserver.w →
        ....

SPEC_app_execute_dependence = □ f: FIREWALL •
        execute.firewall.f → (
                □ w: WEBSERVER • execute.webserver.w → execute.app.ScientificWebApp →
        ....
```

Extracted processes

The collected logs were translated into CSP processes using a simple parser, which converts the individual entries in the log files into CSP events — channels with data. The events were then categorised according to the type, and arranged into a CSP process model. The generic model of the process models obtained is shown here:

```
VM_Extract(T) = start → (⊓ x : T • x ....)
                [|vm_events_interface|]
                OrderEnforcement(T)
vm_events_interface = {|install, execute, config, uninstall, update|}
```

T is the trace of events obtained from a virtual machine, and *OrderEnforcement* is a function that enforces the order of events in T by using timestamps obtained from the log files.

Verification and results

		VM1	VM2
1.	*SPEC_firewall_installed*	pass	pass
2.	*SPEC_firewall_configured*	pass	fail
3.	$SPEC_{a}pp_{i}nstall_{d}ependence$	pass	pass
4.	*SPEC_app_execute_dependence*	pass	fail
5.	*SPEC_app_config_dependence*	pass	pass

Table 6.7: Results of assertions on the specifications

Verification was performed by checking the refinement of the extracted process models against the whitelist. Each process model was abstracted using the abstraction set specified for a given specification and checked for refinement against the specifications in the repository (obtained from the generator implementation described in Section 6.3.6).

The results obtained are shown in Table 6.7. Both VM1 and VM2 had known firewalls installed, *Firestarter* and *Lockit* respectively. They also had the application installed after the firewall and web server were both installed and their web servers were configured appropriately. However, VM2 used unacceptable configurations for the firewall, $Telnet = 1$ instead of $Telnet = 0$, and also executed the web server before the firewall was started. This means that the web server could have been compromised before the firewall was started. Furthermore, even after the firewall was executed, its configurations allowed Telnet traffic to go through, which could lead to attacks against the VM.

6.3.7 Evaluation

In this section, we make a number of observations and evaluate the approach discussed above.

Advantages Over Traditional Whitelisting

Traditional whitelisting-based approach such as that assumed in most trustworthy system architectures check whether an attested software component is listed on an application whitelist. These are restricted to specifications that consider each whitelist entry as an individual entity. The proposed model, on the other hand, is based on a formalism with semantics that facilitate reasoning about concurrency, enabling specification and verification of more powerful properties. For example, a database server might be required to conform to MySQL configuration steps in Listing 9.

```
+ yum install mysql-server mysql-connector-java
+ Configure to UTF-8
    + Modify /etc/my.conf
    + /etc/init.d/mysqld restart
+ run ``mysql'' to create the database and user
    + create database db_example;
    + use db_example;
    + grant all privileges on db_example.* to 'user'@'localhost';
```

Listing 6.1: Example MySQL configuration steps

A traditional whitelisting approach would enable one to verify that the expected version of MySQL was installed and started, but will not that it was restarted after configuring the encoding (UTF-8) or that certain privileges were assigned to a given user. Our approach would allow one to verify that the exact process above has been performed in the required order and with the expected arguments passed to each step. In addition, it provides *counterexamples* upon failure to inform the user about the events that caused it.

Limitations of Common Log Files

During the experiment, we observed that the structure of common log files would only allow a small subset of properties (that are verifiable through our approach) to be verified. For instance, software integrity information is not recorded in the installation log files. To use our approach more effectively, the logging system would have to be configured to capture the evidence specified in Section 6.2.2. In Section 6.4, we discuss how a provenance-based approach helps to address this problem.

Custom Applications

In a collaborative research environment, custom applications may be developed and shared among collaborators. Custom applications as such may not exist in public repositories; hence, conventional installation mechanisms (e.g. package managers) may not be used. To solve this problem, the proposed model can be combined with remote verification [97] to verify the compilation of applications and the whitelist management approach proposed by Huh et al. [72].

Reducing the Model Size

One of the main bottlenecks of using a formal approach is the limitation on the size of the state space that can be handled by existing verification tools. In this case, events collected from the system must be carefully selected according to their importance. For example, during the experiment setup, we decided to restrict the process model to a session so that the lifetime of a virtual appliance can be divided into smaller process models; these can then be verified independently. It may also make sense to collect only those events related to the components in the trusted computing base. As a result of this limitation, the approach suffers from scalability issues. For example, in cases where the behaviour is determined by activities on a collection of systems, then logs would need to be collected from each system, linked together and then verified. However, the use of CSP semantics — more specifically its ability to enable composition of simpler processes into more complex processes using operators such as interleaving — makes it possible to create process models for each of the systems in the collection. These can then be composed together to obtain a process model that captures the activities in the collection.

Expertise Required to Specify Trust-properties

The success of the model has a significant dependence on being able to specify properties correctly and efficiently. Poorly or incorrectly specified properties would render the verification process less meaningful. In this approach the whitelist has been proposed as a repository of specifications, where the specifications capture the sequences of events or combinations of configurations expected of a system. This means that a user specifying the properties has to understand what events or combinations of events should be considered as leading to trustworthy behaviour. This problem can, however, be eased by developing communities dedicated to collecting and characterising events. Similar efforts can be observed from the work on creating catalogues

of vulnerabilities [168], platform configurations [102] or attack patterns [167]. Alternatively, the whitelist management approach presented by Huh et al. [72] could be modified so that rather than sharing the configurations or expected integrity values, the specifications could be shared instead.

6.3.8 Summary

Properties that fall in the shadow of the future class require collecting information from an operational system and treating this information as evidence to support the reasons why a system could be expected to behave in some expected manner. In this section, we have proposed a trace-based verification model as a mechanism for collecting and creating runtime abstractions of the system that can be verified against properties of interest. The event logs are translated into a process model, which is, in turn, verified against properties captured in the repository of specifications — the whitelist. The verification results indicate whether or not the process model could have performed events that are not permitted by the specifications.

Unlike the traditional whitelisting approaches, this approach enables complex properties, including concurrency, order and dependency, to be modelled and checked. However, such properties require some expertise in identifying and specifying which sets of events or combinations of them represent trustworthy behaviour. Furthermore, the model requires careful abstractions to limit the size of the state space. This has an effect on the scalability of the model. In the next section, we discuss how a provenance-based approach could be used to complement this approach in order to eliminate some of the limitations identified for this model.

6.4 Provenance-based Verification Model

In Section 6.1 we discussed the need to collect comprehensive evidence about a system's ability to satisfy trust-properties of interest. In the previous section, we demonstrated one approach towards collecting and using this evidence to reason about trust-properties. The approach involves collecting traces of events from a system and translating them into a process model that can be verified against trust-properties captured through specifications in the whitelist. However, considerable effort and expertise is required to effectively and correctly capture the properties of interest. Furthermore, if more details are included in the logs, then more abstractions have to be performed to reduce the state space of the models. This could result in the loss of valuable data for use in trust decisions.

In this section, we argue that provenance provides more comprehensive evidence that can be used to reason about trust-properties without compromising the ease of property specification and scalability. Towards this end, and motivated by the realisation that trusted computing and provenance seek to address similar issues [99], we propose a *provenance-based model* in which activities on a system are captured as a *provenance graph*. The model extends the Open Provenance Model (OPM) [115] to enable provenance to be captured in a manner that supports verification of trust-properties. The approach has several advantages, some of which include:

1. ability to provide a more comprehensive view of a system — resulting from the use of an enhanced provenance-model;

2. support for interoperability — resulting from the use of standardised semantics;

3. support for varying complexity in the types of properties that can be verified — as some properties can be expressed using queries over the provenance-graph while other more complex properties can still be verified by transforming the graph into process models; and

4. support for multiple abstractions — enabled by the use of graph semantics that allow extraction of sub-graphs that concentrate on different aspects of evidence.

In the next section, we describe the proposed enhancements to the provenance-graph. This is followed by a description of how this graph is used to reason about trust-properties in Section 6.4.2. We demonstrate how this approach works on an experiment with the trustworthy collaborative system.

6.4.1 Graph-based Evidence Representation

We propose a provenance-based model that extends the open provenance model (OPM) [115] to capture the evidence. In this model information about the data used, processes performed, entities that perform these processes and any new data generated is captured as a set of RDF triples [24], where each triple (X, Y, Z) specifies that a component X was related to another component Z through the property Y. In the rest of this section, we describe the extensions to OPM necessary to support reasoning about trust properties.

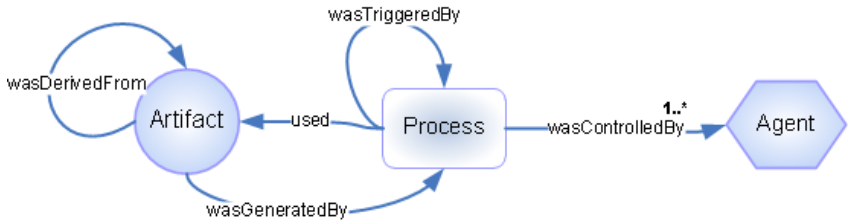


Figure 6.3: Th main components (Artifacts, agents and processes) of OPM

A Summary of OPM Semantics

Since our model builds on OPM, we begin with a summary of the main semantics of OPM, a detailed discussion of which can be found in Groth and Moreau [115].

OPM defines three main entities in a provenance record. These include: *Agent*, *Artifact* and *Process*, where an agent is an entity capable of performing a process, an artifact is an immutable piece of state and a process is a series of actions that use artifacts and generate new artifacts. These entities are related through a number of properties as depicted in Figure 6.3. The *wasTriggeredBy* (WTB) relationship defines a relationship in which one process is made operational by another process. A process can be specified to have been controlled by multiple agents through the *wasControlledBy* (WCB) property. Artifacts used in a process are indicated through the *used* property while those that are created by a process are related to the process that created them through the *wasGeneratedBy* (WGB) property. The *used* and *WGB* properties must occur after the process has been created. To maintain the link between those artifacts that are used and those created, the *wasDerivedFrom* (WDF) property is used to specify that one artifact was derived from another. However, these semantics introduce some limitations (as discussed in the following sections) towards capturing the evidence for the purpose of reasoning about trust-properties.

Program Execution

A program can be captured as an agent in OPM. However, the semantics of the properties defined in OPM limit the ability to express execution relationships among programs. Consider the boot phase of a system in which the BIOS executes the boot loader, which in turn executes the operating system kernel. Using OPM, this phase can be captured as illustrated in Figure 6.4. The BIOS controls a process, $P1$, which triggers another process, $P2$, controlled by the boot loader. $P2$ triggers $P3$, which is controlled by the operating system kernel. The semantics of the *WCB* property, however, imply that a single process can in fact be controlled by multiple programs

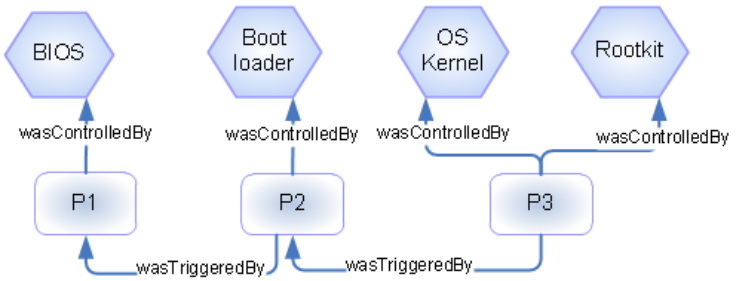


Figure 6.4: Using OPM to capture the boot phase

(e.g. *P3 wasControlledBy* OS Kernel and Rootkit in Figure 6.4). Alternatively, the concept of *role* defined in OPM could be used to specify the role played by each program linked through the *WCB* property. However, roles are defined as labels and would still require a similar effort in defining semantics to make them useful in trust evaluation.

An alternative approach would be to use the TCG integrity architecture to create a chain of trust which captures the notion that a program in the chain executed and transferred control to the next program in the chain. However, such an approach is not sufficient to capture the idea that other activities could have been happening at the same time as the execution of programs in the chain. To overcome these limitations, we propose an extension to OPM that captures aspects about program execution. The extension, illustrated in Figure 6.5, enables programs to be related to the processes through which they are executed and the component executing them. This has an advantage that when establishing trust not only is the resulting chain of execution checkable, but also the processes in which the chain was created. We formally define the extension to include *ExecutionProcess* and *BootstrapProcess* processes and a number of properties that relate programs to the processes through which they were executed as well as to the program that initiated the execution (note: *ran* is a function which returns the range of a given relation).

$$[Node, Time]$$

$$Process, Agent, Artifact : \mathbb{P}\, Node$$
$$Program : \mathbb{P}\, Node$$
$$\hline$$
$$Process \cap Agent = \emptyset$$
$$Program \subseteq Agent$$

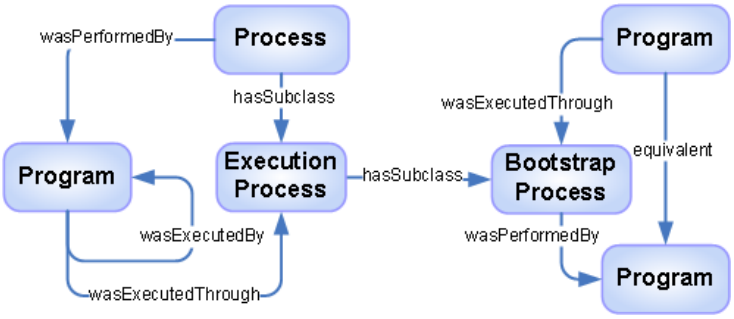


Figure 6.5: Structure of an extension to support program execution

$$
\begin{array}{|l}
wasPerformedBy : Process \pfun Program \\
SystemProcesses : \mathbb{P}\, Process \\
SystemPrograms : \mathbb{P}\, Program \\
ExecutionProcess : \mathbb{P}\, Process \\
wasExecutedThrough : Program \pfun Process \\
wasExecutedBy : Program \pfun Program \\
wasExecutedAt : Program \pfun Time \\
BootstrapProcess : \mathbb{P}\, Process \\
\hline
ExecutionProcess = \operatorname{ran} wasExecutedThrough \\
\operatorname{dom} wasPerformedBy = SystemProcesses \\
\operatorname{dom} wasExecutedThrough = SystemPrograms \\
\operatorname{dom} wasExecutedBy = SystemPrograms \\
\operatorname{dom} wasExecutedAt = SystemPrograms \\
\forall\, p1, p2 : Program \bullet \\
\quad wasExecutedBy\ p2 = p1 \Leftrightarrow (\exists\, e : ExecutionProcess \bullet \\
\qquad wasPerformedBy\ e = p1 \land wasExecutedThrough\ p2 = e) \\
\forall\, e : ExecutionProcess \bullet \\
\quad e \in BootstrapProcess \Leftrightarrow (\exists\, p1 : Program \bullet \\
\qquad wasPerformedBy\ e = p1 \land wasExecutedThrough\ p1 = e)
\end{array}
$$

Intuitively, an *ExecutionProcess* is performed by a specific program and yields another program (i.e. the new program goes into the running state). A special type of execution process in which the program that performs the execution is the same as the program that is yielded from this process is referred to as a *BootstrapProcess*.

Program Configuration

Configuration settings play an important role in determining the behaviour of a program. Grawrock [59] notes that in any non-trivial system, there will be a number of

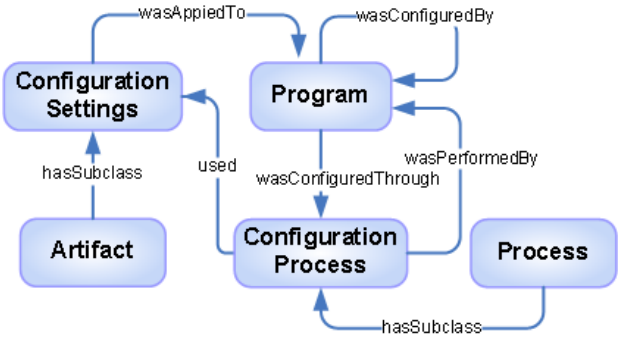


Figure 6.6: Structure of an extension to support program configuration

configuration options that may affect how a system behaves. For this reason, configuration settings used in a system must be considered when evaluating the trust-properties.

In OPM, configuration settings can be captured as a type of *Artifact*. This artifact can then be linked to the process that uses it through the *used* property to capture the idea that a process is configured with the configuration settings specified. However, as discussed in the previous section, the semantics of *WCB* imply that configuration settings used by a process cannot be linked to the specific program being configured because multiple programs are linked to the same process that uses the artifact. Furthermore, the semantics of *used* imply that a process has to start its operation *before* it can be configured. However, for trust evaluation it is important to capture the idea that a program was configured in a certain way before it engaged in some other activities. To achieve this, we define an extension to OPM in which *ConfigurationSettings* is defined as a type of artifact that can be used to configure a program in a process called *ConfigurationProcess*. We capture this extension formally as follows, also illustrated in Figure 6.6.

$$
\begin{array}{l}
ConfigurationSettings : \mathbb{P}\, Artifact \\
ConfigurationProcess : \mathbb{P}\, Process \\
used : Process \leftrightarrow Artifact
\end{array}
$$

$$
\begin{array}{|l}
wasAppliedTo : ConfigurationSettings \leftrightarrow Program \\
wasConfiguredBy : Program \leftrightarrow Program \\
wasConfiguredThrough : Program \leftrightarrow ConfigurationProcess \\
\hline
\mathrm{dom}\ wasConfiguredBy = SystemPrograms \\
\mathrm{dom}\ wasConfiguredThrough = SystemPrograms \\
\forall\, p : ConfigurationProcess \bullet \\
\quad (\exists\, c : ConfigurationSettings \bullet c \in used(\!|\ \{p\}\ |\!)) \\
\forall\, c : ConfigurationSettings;\ p : Program \bullet p \in wasAppliedTo\,(\!|\ \{c\}\ |\!) \Leftrightarrow \\
\quad (\exists\, e : ConfigurationProcess \bullet c \in used(\!|\ \{e\}\ |\!) \wedge \\
\qquad e \in wasConfiguredThrough(\!|\ \{p\}\ |\!)) \\
\forall\, p1, p2 : Program \bullet p1 \in wasConfiguredBy(\!|\ \{p2\}\ |\!) \Leftrightarrow \\
\quad (\exists\, e : ConfigurationProcess;\ c : ConfigurationSettings \bullet \\
\qquad wasPerformedBy\ e = p2 \wedge p1 \in wasAppliedTo\,(\!|\ \{c\}\ |\!))
\end{array}
$$

This extension allows us to capture properties such as "a program X configured another program Y with settings Z".

Integrity Measurement

Integrity measurement can be captured as a process using OPM, so that the entity being measured is linked to the integrity measurement process through the *used* property while the resulting integrity value is linked to the process that performs the measurement through the *WGB* property. However, the semantics of *used* imply that only artifacts can be integrity measured because the used relationship can only be applied to artifacts. To overcome this limitation, we introduce an extension to OPM, illustrated in Figure 6.7 and formally defined below, which includes a type of process referred to as *IntegrityMeasurementProcess*, an artifact called *IntegrityValue*, a *Measurable* type and a number of properties that relate these concepts (and those already defined in OPM). *Measurable* is defined as a union between agents and artifacts to capture the idea that both agents and artifacts can be integrity measured.

$$
\begin{array}{|l}
performedOn : IntegrityMeasurementProcess \rightarrow Measurable \\
integrityOf : IntegrityValue \nrightarrow Measurable \\
measured : Program \leftrightarrow Measurable \\
\hline
\forall\, p : Program;\ m : Measurable \bullet measured\ p = m \Leftrightarrow \\
\quad (\exists\, e : IntegrityMeasurementProcess \bullet performedOn\ e = m \\
\qquad \wedge\ performedBy\ e = p)
\end{array}
$$

The extension specifies that *IntegrityMeasurementProcess* is a process that can take agents, in addition to artifacts, as input and produce another artifact of type

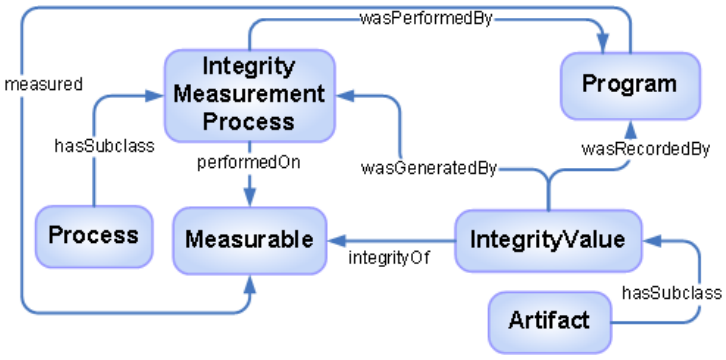


Figure 6.7: An extension to support capturing integrity measurements using OPM

IntegrityValue. The entity whose integrity is being taken can be linked to the resulting integrity value through the *integrityOf* property and to the *IntegrityMeasurementProcess* through the *performedOn* property.

Communication

Components on a system communicate through various means to provide services to one another (e.g. remote procedure calls), inform each other about their activities or observations (e.g. events, message broadcasts) and exchange information for use in computations. This communication creates interactions which determine the flow of information within and across systems.

To cater for communication aspects of the system, we make use of a combination of the D-profile proposed by Groth and Moreau [63] and the common module defined for the open provenance model vocabulary (OPMV)[2]. The D-profile defines the relationship between a sender process and the message it sends as well as the receiver process and the message it receives. This is useful for capturing communication. However, if we need to capture interactions resulting from this communication, the D-Profile falls short. To solve this, we complement it with concepts defined in the common module. More specifically, we use *Download*, *downloadUri*, *connection* defined in the namespace, *http://purl.org/net/opmv/types/common#* and *Connection* defined in the namespace, *http://www.w3.org/2006/http#*.

[2]Common Module is a specialisation of OPMV that defines commonly used terms that are not defined in the OPM specifications; see http://code.google.com/p/opmv/wiki/GuideOfCommonModule for details

Assertion Model

We have so far defined a way of capturing the various kinds of evidence as provenance statements. However, these statements are only sound if they can be traced to a *root of trust*. Groth and Moreau [63] note that in distributed systems, where there may be multiple monitoring and reporting components, each provenance entry must be linked to the entity that reports it. They introduce the *attributedTo* property to link an account of provenance to an entity responsible for it. However, the soundness of these attributions can only be determined if a link can be created between the attributing entity (*assertor*) and a root of trust, i.e. a root of trust for assertion.

To achieve this, we propose an extension to OPM, called *Assertion Model*, which makes use of the earlier defined extensions to create links between the assertor and other components which could potentially serve as roots of trust. In other words, this can be used to create multiple chain of trust (based on different aspects of the system, not just the measure-before-load [59] as is the case for TCG based chain of trust) with the assertor at one end of the chain and a root of trust for assertion at the other end.

In this model, each tuple (id, s, p, o) is referred to as *Assertion*. Each assertion is linked to an *assertor* through the *asserts* relation and to some *AssertionMetaData*. The assertion metadata records the integrity of the entities in an assertion as well as that of the assertor and must be supplied by an agent that is either implicitly trusted (i.e. is a root of trust) or explicitly trusted (i.e. through integrity measurement). This allows us to specify that an agent A asserts that a subject s is related to an object o through a property p and also to support this through collection of assertion metadata from some trusted monitor.

$$Property : Node \leftrightarrow Node$$
$$ImplicitOrExplicitTrusted : \mathbb{P}\, Agent$$

$$[Identity]$$

Assertion

$$uid : Identity$$
$$subject, object : Node$$
$$relation : Property$$

AssertionMetaData

$$uid, assId : Identity$$
$$subjectIntegrity, objectIntegrity, assertorIntegrity : IntegrityValue$$
$$assertedAt, monitoredAt : Time$$

$$
\begin{array}{|l}
asserts : Agent \leftrightarrow Assertion \\
monitors : Agent \leftrightarrow AssertionMetaData \\
\hline
\forall\, assertor : Agent;\ ast : Assertion \mid (assertor, ast) \in asserts \bullet \\
\quad (\exists\, AssertionMetaData;\ monitor : Agent \bullet \\
\qquad (monitor, \theta AssertionMetaData) \in monitors \land \\
\qquad integrityOf\ ast.subject = subjectIntegrity \land \\
\qquad integrityOf\ ast.object = objectIntegrity \land \\
\qquad integrityOf\ assertor = assertorIntegrity \land \\
\qquad ast.uid = assId \land monitor \in ImplicitOrExplicitTrusted)
\end{array}
$$

6.4.2 Reasoning about Trust-properties

The graph representation discussed above provides a means of representing the evidence about the activities on a system. But how useful is this evidence and how does the graph representation help in reasoning about trust-properties? This section addresses these questions by proposing an approach in which the evidence is validated against a set of criteria which aim to determine its soundness before verifying it against trust-properties of interest.

Evidence Validation

The evidence presented in a provenance graph can come from multiple sources. As discussed in Section 6.4.1, for the purpose of reasoning about trust-properties, this evidence must be linked to the entities that generate it. Therefore, we consider evidence to be valid if it can be linked to an entity that can be securely identified. The use of the assertion model makes this possible by providing a link between the assertions and the assertors so that validation is based on the ability to securely identify assertors and monitors of assertions, using the *asserts* and *monitor* relations, respectively.

Property Specification and Verification

Our verification model is based on RDF graph pattern matching provided in SPARQL [136]. First, each property specification, discussed in the previous section, is captured as a basic graph pattern (BGP) — a set of triples which may have some of the elements represented by variables. Then the obtained BGP is mapped to the graph (or a sub-graph) to determine if there is an entailment relationship between the BGP and the graph. In the remainder of this section, we discuss how properties are specified and verified on a given graph.

- *Presence/Absence of Triple Patterns*: properties can be specified in terms of the presence or absence of certain triples. For example, to specify that a firewall was installed, one can check the graph to determine whether or not the triples *(installProcess, wasPerformedBy, rpm)* and *(installProcess, used, firewall.rpm)* exist. This is achieved by specifying the triples to be checked as a BGP (when the values of the triple elements are important) or graph templates (when certain values can be ignored). Verification is achieved by performing a query in the form of *ASK*, which returns true or false, depending on whether the specified triples can be found in the graph. Listing 6.2 shows how the example of firewall installation can be verified.

```
ASK
        { :installProcess :wasPerformedBy :rpm . }
        { :installProcess :used :firewall.rpm . }
```

Listing 6.2: Example query for determining presence/absence of triples

- *Values of Triple Elements*: elements of a triple have values which can be used to infer certain information about the behaviour of a system. In the TCG-based integrity mechanism, for instance, trust is based on the presence of components with known integrity values. We support specification of triple element values using the FILTER feature of SPARQL. For example, to determine that in a given execution, the installed firewall had a certain integrity value, a query such as that shown in Listing 6.3 can be specified.

```
ASK
        { :installProcess :used  :firewall.rpm .
          ?iv   :integrityOf :firewall.rpm .
        FILTER ( ?iv = "cdf84324"^^xsd:string )}
```

Listing 6.3: Example query for determining triple element values

- *Supporting Multiple Abstractions*: the use of graph semantics allows any set of triples to be defined as a graph. This enables us to provide multiple abstractions, so that a sub-graph that concentrates on certain types of assertions can be obtained from a provenance graph by selecting a set of triples that satisfy some desired properties. This is achieved by specifying a graph template that includes properties useful for a certain abstraction. For example, to capture a graph that can be used to determine an execution chain of trust on a system, a graph template such as '*?x :wasExecutedBy ?y*' can be used to return

all triples that have a *wasExecutedBy* property between any two entities (represented by the variables x and y). Such a graph can be obtained by using the *CONSTRUCT* query form on a provenance graph, which returns a graph matching the triples specified. For example, the query in Listing 6.4 returns a graph which only includes assertions related to programs executed on a system. The resulting graph can be subjected to further analysis as discussed above.

```
CONSTRUCT { ?x ?p  ?y .}
WHERE { ?x :wasExecutedBy ?y .}
```

Listing 6.4: Example query to create abstraction for executions

- *Sequencing of Triples*: in most cases, a triple taken in isolation does not provide much information. To develop a more meaningful judgement of the behaviour of the system requires a way of relating the triples. One such relationship is the sequencing of triples. For example, to specify that a given program was configured in a certain way before it participated in a process, X, would involve checking that the program configuration occurred before X was performed. Triple sequencing can be specified using the assertion model, where each assertion is linked to the time instance at which a particular triple occurred, using the *occuredAt* property. Given a sequence of triples $T =< t1, t2, ...tn >$, the FILTER construct can be used to relate the times at which each triple occurs. Listing 6.5 shows an example (s = subject, p = property and o = object).

```
ASK
        {t1.s   t1.p    t1.o .
        t1.(s,p,o)  :occuredAt   ?x1 .
        t2.s   t2.p    t2.o .
        t2.(s,p,o)  :occuredAt   ?x2 .
        ...
        tn.s   tn.p    tn.o .
        tn.(s,p,o)  :occuredAt   ?xn .
        FILTER ( ?x1 < ?xn <...<?xn )}
```

Listing 6.5: Example query to determine sequence of triples

6.4.3 Verifying Virtual Appliances in a Collaborative System

In this section, we describe an experiment that demonstrates how the provenance-based model can be used to verify trust-properties of platforms before they can be used in experiments.

Trust Properties

We discussed earlier that virtual appliances could be intentionally or unintentionally mis-configured or have malware installed on them and thus researchers need to verify the state of the VAs before using them for the experiments. Below, we identify some examples of trust-properties that could be verified towards achieving this goal.

1. *VA-TP1: all packages installed on a VA were authentic* — packages installed on a VA will provide the services for which a VA was intended and, therefore, these packages have a significant impact on the behaviour of a VA.

2. *VA-TP2: known configurations were applied to programs executed as part of the VA build process* — the behaviour of programs used in the build process for a VA not only depend on the integrity of the executable code but also the correctly or configurations applied to them. For this reason, it is important that the configurations used are identified and verified.

3. *VA-TP3: known scripts will be executed when VA is deployed* — a VA can be configured to run certain scripts when ti is launched. These scripts can be used to achieve a number of things including resetting credentials such as SSH keys, updating installed packages, setting database configurations and so on.

In the following sections, we demonstrate how these properties were specified and verified against a graph generated from provenance collected during the build process.

Set-up and Provenance Collection

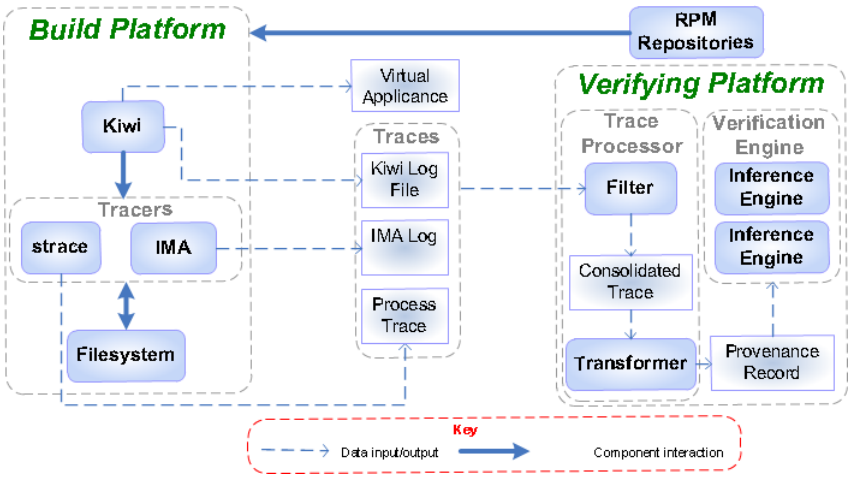


Figure 6.8: An experimental setup for generating provenance for virtual appliances

The experiment was set-up as illustrated in Figure 6.8. A build platform was setup using openSUSE 11.3 running on a kernel compiled with IMA support. Kiwi imaging system[3] and *strace* [29] packages were installed and a simple shell script was set-up to execute Kiwi using *strace* as a tracing tool. Execution traces were collected including log files from Kiwi, integrity measurement log (an extract of which appears in Listing 6.6) and a trace generated by *strace* (an extract appears in Listing 6.7), which were processed and verified on a separate platform.

```
10 000000 ima 000000 boot_aggregate
10 7ba06b ima 095baf /init
...
10 ba0922 ima bb4476 ./build.sh
10 f80069 ima e54a84 /usr/bin/strace
10 f97bd8 ima e10ec0 /usr/sbin/kiwi
....
10 1125be ima 908990 config.xml
10 b97636 ima 185cb1 KIWIConfig.sh
....
10 6b81fa ima cc622c /usr/bin/zypper
....
10 988080 ima bddd59 zypper.conf
10 229eb6 ima 6bf998 opensuse.org_distr_xx_.repo
...
10 2d6856 ima 3b2310 openSUSE−xx−.i586.rpm
10 e828ee ima b0d515 filesystem−xx−.i586.rpm
10 827556 ima bc4d2a vim−base−7.xx.i586.rpm
....
10 d9545c ima f56808 config.sh
```

Listing 6.6: IMA log with hash values truncated to six digits

```
3056 18:19:08 clone(child_stack=0,....) = 3057
3057 18:19:08 execve("/usr/bin/zypper", ....) = 0
...
3057 18:19:42 clone(child_stack=0,....) = 3069
3069 18:19:42 execve("/bin/rpm",..,http:download.
    opensuse.org..openSUSE−xx−.i586.rpm
...
4354 18:21:51 open("/tmp/prov-va/tmp/config.sh",..
```

Listing 6.7: Example entries in strace output

Graph Representation

A provenance graph that conforms to the model described in Section 6.4.1 was generated from the collected traces. The integrity measurement log provides integrity values of the components on the platform and therefore enables us to create instances of *IntegrityValue* and specify the *integrityOf* property. The trace generated by *strace* provides information about relationships among the programs executed and the data they use.

[3]Kiwi is a tool, from openSUSE Build Services, used for creating VM images.

A slice of the resulting graph is shown in Figure 6.9. The graph shows how programs and data used in various processes that occurred on a VA are related and how IMA provides assertions about integrity values.

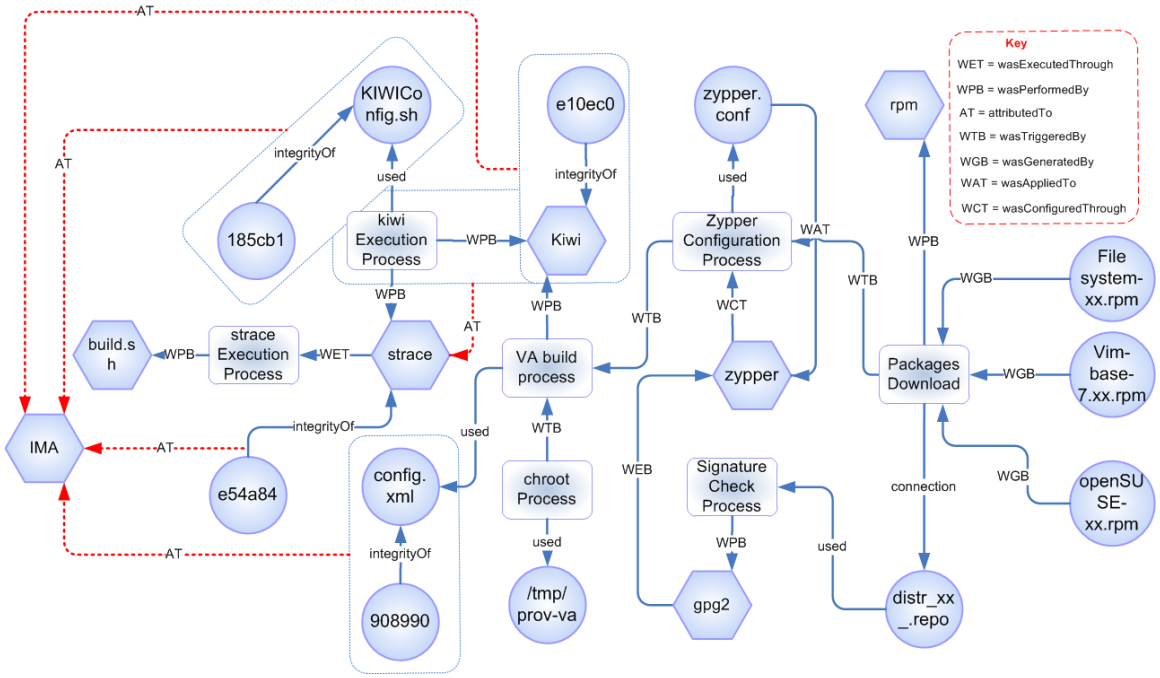


Figure 6.9: A provenance graph — showing only a small number of entities

Verifying VA-TP1

In our experiment, the authenticity of packages can be achieved in two ways: check the integrity values recorded for each of the"*.rpm*" packages or determine whether they were obtained from a trusted repository. In the first option, the packages are filtered so that only ".rpm" packages are included. Then each package that was generated by some process, such as *Download*, is linked to its recorded integrity value. Listing 6.8 shows an example query used to verify this property and the results are shown in Listing 6.9.

The second approach relies on trusting the repository from which packages have been installed. This is particularly useful in cases where the behaviour and content of the repository system has been certified by some independent party trusted by researchers. In this case verifying that all packages were downloaded from a trusted source would be sufficient to verify this property. Listing 6.10 shows a query used to check the source of packages and the results obtained are shown in Listing 6.11.

```
SELECT {?pkg ?iv }
WHERE {?pkg  :wasGeneratedBy ?x .
        ?iv  :integrityOf  ?pkg .
        FILTER regex(?x,"rpm")}
```

Listing 6.8: Query to determine the integrity of packages installed

```
Artifact    IntegrityValue
"filesystem-xx-.i586.rpm" "b0d515"
"vim-base-7.xx.i586.rpm" "bc4d2a"
"openSUSE-xx-.i586.rpm"  "3b2310"
```

Listing 6.9: Results of package integrity query

```
SELECT {?pkg ?y }
WHERE {?x :connection  ?y .
        ?xrdf:type :Download .
        ?pkg :wasGeneratedBy ?x .}
```

Listing 6.10: Query to determine the source of packages installed

```
Artifact      Connection
"filesystem-xx-.i586.rpm" opensuse.org_distr_xx_.repo
"vim-base-7.xx.i586.rpm" opensuse.org_distr_xx_.repo
"openSUSE-xx-.i586.rpm"  opensuse.org_distr_xx_.repo
```

Listing 6.11: Results of package source query

Verifying VA-TP2

This can be achieved by checking the integrity values of the configuration settings that have a *wasAppliedTo* relationship with programs by using a query such as the one shown in Listing 6.12. This query returns the integrity, x, of configuration settings, $?y$, used in a configuration process in which a program z was configured.

```
SELECT { ?y ?x ?z}
WHERE { ?x :integrityOf ?y .
        ?p rdf:type :ConfigurationProcess .
        ?p :used ?y .
        ?z :wasConfiguredThrough ?p .
        ?y :wasAppliedTo ?z}
```

Listing 6.12: Example query to check configurations

```
ConfigurationSettings IntegrityValue Program
"zypper.conf"     "bddd59"  "zypper"
"KIWIConfig.sh"    "185cb1"  "Kiwi"
```

Listing 6.13: Results of checking program configurations

Verifying VA-TP3

In the build process, start-up scripts are copied to specific locations on the file system of a VA. Therefore, this property can be accomplished by checking the integrity of the programs or artifacts that have been placed in a certain location. Listing 6.14 shows a query that determines the integrity of artifacts that have been copied to the "/etc/init.d" directory.

```
SELECT {?x ?y }
WHERE {?x rdf:type :artifact .
        ?y :integrityOf ?x .
        FILTER regex(?x, "/etc/init\.d") }
```

Listing 6.14: Query to determine the integrity of scripts that will be executed

```
Artifact  IntegrityValue
"config.sh"  "3de324"
"image.sh"  "359aa3"
```

Listing 6.15: Results of checking scripts copied

6.4.4 Evaluation

Interoperability and Extensibility

Our model can be extended with semantics useful for a given application domain by defining new concepts or extending existing concepts. The new concepts can then be linked to concepts that exist in the model or to other new ones by defining or using existing properties. For example, entities in the TCG schemas can be mapped to either programs or artifacts to take advantage of the interrelations among components and thus enable a more comprehensive verification of platform configurations.

Collecting and Securing the Evidence

One key issue with the use of provenance in verifying trustworthiness is establishing the trustworthiness of the provenance itself. Considerable effort [62, 65] has been directed towards this end and it is not our intention to provide a solution for this problem. Instead, we have assumed that this information is secured and concentrated on developing a model that enables one to use this information to verify trust-properties. The evidence can further be tagged with trust values to indicate the belief that the assertor has in the assertions [65], allowing quantitative measurement of trust.

Assumptions on Infrastructure

In the experiment, we considered trust-properties that can be established through evidence obtained from the build platform. However, there are other aspects that could affect the behaviour of a virtual appliance when launched. For example, two identically configured VAs could behave differently if the runtime parameters passed from the hypervisor are different. We assume that the hypervisor would launch all VAs with identical parameters. Future work may involve investigating the kinds of properties that are applied to VAs at runtime and how such parameters affect the behaviour of VAs.

6.4.5 Summary

The nature of evidence for use in trust evaluation and how it can be represented and used is still an open problem. Existing mechanisms such as the TCG integrity schema are limited to a specific aspect of a system's operation (e.g. chain of program execution). We have proposed a provenance-based model in which evidence is represented as a provenance graph that captures activities that occur on a system. This model specifies relationships among system components and data to enable evaluation

against trust-properties that fall into the shadow of the future class. Our application to virtual platforms for use in a collaborative system demonstrates that the approach enables verification of more comprehensive properties.

6.5 Discussion

6.5.1 Satisfying the requirements

In Section 3.7.3 we identified a number of requirements which must be satisfied by a model that can be used for reasoning about *shadow of the future* properties. We now discuss how the approach proposed in this chapter meets those requirements.

`REQ 10` specified the need to collect information that could serve as reliable attestations. This requirement is satisfied by recording activities on the build platform or from a running virtual machine either as a trace or events or as provenance. For `REQ 11`, we assumed that a trustworthy evidence collection system is available. For example, a trustworthy logging system proposed by Huh and Martin [73] could be used. `REQ 12` is met by making the evidence accessible to those wishing to rely on the systems, such as researchers. Finally `REQ 13` is met by using the trace analysis approach that supports generation of the repository of specifications and process models of the virtual appliance events or a provenance-based model that represents this evidence as a graph. Both models enable properties to be specified in a manner suitable for verification against the underlying model.

6.5.2 Reliance on Collection Mechanisms

The approach proposed in this chapter relies on there being evidence that has been collected from the system. The trace-based approach relies on the existence of a trustworthy logging component. However, log files have a number of limitations in terms of the structure of evidence contained within. Though the provenance-based model improves on this situation, there aren't readily available provenance collection systems capable of providing such evidence in a secure manner. For example, we experimented with the Provenance-Aware Storage System (PASS) [117] prototype implementation. Using PASS, we specified a provenance-aware volume to capture provenance generated during execution of a given task. Unfortunately, the current PASS prototype has dependencies on lower versions of some libraries, e.g. *udev*, which makes it impossible to work with later versions of the kernel. Furthermore, security is not the primary concern of PASS.

6.5.3 Scalability

The reliance on availability of evidence, discussed in the previous section, also implies that the more comprehensive the evidence that can be collected from the system is, the more comprehensive property verification becomes. However, this raises scalability issues. First, increasing the amount of evidence corresponds to an increase in the expectations on the capabilities of the relying entities. For example, they must have capacity to process large amounts of data. Second, the more evidence is collected, the more storage capacity will be required. In some circumstances such as the collaborative scenario discussed earlier, it might be necessary to keep a copy of the evidence used. For example, researchers may want to demonstrate that their experiments were performed on an environment that was verified for certain properties. In this case, the evidence used in the verification may have to be kept.

6.5.4 Generalisation of the Approach

Although the approach has been applied to verification of virtual appliances, it is not limited to such an environment. Indeed, other areas might benefit from this approach. For example, the notion of having a repository of specifications can be applied to any application domain that requires verification of the observed traces of events against some desired properties. For example, events captured in a stored measurement log during the trusted computing authenticated boot process could be translated into a process model and used to verify a chain of trust against properties such as *measure before load*, *integrity and order of application execution*.

The approach can also be used to verify events observed from processes that work together towards a common goal. For example, events observed from members of a trusted virtual domain [61] can be modelled as processes that work together to enforce a common policy, and verified against the specifications in the repository or as a provenance graph that can be queried for specific properties.

6.5.5 Comparison with Related Work

The work presented in this chapter builds on and provides alternatives to some existing research. The idea of verifying trust-related properties using log analysis has been investigated by Ulrich et al. [169], who proposed a tracing-based analysis system for checking security requirements of a distributed system using event logs at runtime. In their approach, event logs are processed into a specification of a system, which in turn, is checked against predefined security requirements using the SDL

model checker. Their approach, however, does not consider properties resulting from concurrent execution of components. Furthermore, the properties verified are arbitrary, whereas in our approach we relate the properties to specific aspects of a system architecture that could influence the behaviour of a system.

Aalst et al. [171] have developed an algorithm for extracting a process model from a workflow log and a method for representing the model in terms of Petri nets. Here, the workflow log contains information about the workflow process as it is being executed. Something like their approach, in our opinion, could be extended and used together with our approach to construct the process models representing the virtual machine executions.

Having realised the complexity of correctly configuring virtual infrastructures to conform to enterprise security policies, Bleikertz et al. [17] have described an automated virtual infrastructure validation and assurance system. Their system unifies the configurations of virtualisation infrastructures (e.g. virtual machines) into a generic data model. This data model is then visualised through a data flow graph, which in turn, is used to detect any breach of security requirements.

Our work on provenance was motivated by Lyle and Martin [99] who note that provenance and trusted computing could complement each other, and builds on the semantics of open provenance model (OPM) proposed by Moreau [115]. Similar ideas of using provenance for security analysis have been investigated by Wei [79]. Our work differs in that we are interested in the evaluation of trustworthiness of a system using the generated provenance rather than the trustworthiness of provenance records.

The idea of developing a more comprehensive view of the system configurations has also been discussed in [133, 153]. Presti [133] proposes the notion of a tree of trust as a mechanism for representing verification data. Schmidt et al. [153] builds on this structure and proposes modifications to the TPM command set and data structures to support the derivation of tree-formed verification data. In our case, the TPM does not require any modifications. Instead it is simply used to validate the authenticity of the nodes in the graph. Furthermore, our graph-based approach provides a richer semantics — enabling verifiers to consider multiple paths to a node — whereas the tree structure proposed by Presti [133] only provides a single way of getting to a particular node. In addition, our model is extensible and supports interoperability.

The TCG Infrastructure Working Group recognises that trust establishment must consider the origin, condition and history of components used to construct the platform. However, the proposed architecture [163] is limited to capturing components that exist on a system rather than how those components interact or how they are

related. Our approach is more comprehensive in that it includes the activities that occur on the system in question and relations among components involved in those activities.

6.6 Conclusion

Properties that fall in the *shadow of the future* class are concerned with identifying whether or not the current state of a system could lead to future trustworthy behaviour. To establish this, information about the activities that have occurred on the system including processes performed, programs executed, configurations applied, data used and so on, needs to be collected from the system and represented in a manner suitable for verification of these properties. This chapter has proposed two complementary models for capturing the runtime abstractions of the system and using these abstractions for reasoning about trust-properties. A trace-based verification model is used to translate a trace of events on a system into a process model that is then verified using a formal approach. While this model is found to support verification of complex properties, it requires significant effort and expertise to *correctly* capture the intended properties. Therefore, a provenance-based approach is used to complement the trace-based approach by enabling specification of properties in the form of queries performed over an enhanced provenance graph.

Chapter 7

Application to Trust Domains: Mergers and Acquisitions Scenario

The three previous chapters covered models and associated frameworks that can be used to enable reasoning about trust-properties that can be guaranteed by a given system. The models have been demonstrated on individual scenarios, each emphasising a different aspect of a system architecture. However, the question of how these models can be incorporated into a system and the benefits to be gained still remains.

This chapter seeks to address these questions by demonstrating the applicability of the proposed approach to *trust domains*. The idea is that the notion of a trust domain, defined in the next section, used to enable secure information sharing, requires mechanisms that enable relying parties (i.e. those involved in a trust domain, contributing some data to it or monitoring it) to understand the level of trust in a trust domain.

Using a scenario from the financial industry involving a mergers and acquisitions (M&A) transaction, we demonstrate that the approach presented in this dissertation can form the foundations of a trust domain. We achieve this by demonstrating that: i) the proposed approach could be incorporated into an architecture for trust domains; ii) the architecture-based framework proposed in Chapter 3 could help classify and analyse trust-properties of a trust domain; and iii) the models and associated frameworks proposed in Chapters 4, 5 and 6 could be applied to enable reasoning about identified trust-properties. We evaluate the advantages and challenges of this approach in Section 7.5 before giving conclusions in Section 7.6.

7.1 Trust Domains: Enabling Secure Information Sharing

The need to share information can be identified in many scenarios. However, the sensitive nature of certain information could have a significant impact on how, with whom and when it can be shared. This could: i) prevent or discourage participants from sharing it; ii) limit the amount of it that can be shared; and iii) increase the time taken between when information is requested to when it is provided.

To overcome these challenges, the Trust Domains project[1] aims to develop a framework that can be used to provide mechanisms that enable secure information sharing while making the level of security that can be achieved using these mechanisms explicit. This is achieved through the notion of a *trust domain*, defined as *the state and processes that allow resources to be shared between entities that are members of the domain and where these entities have an expectation of and exhibit shared and predictable behaviour to protect the resources.*

A trust domain provides a foundation for securely sharing information among a group of (possibly distrusting) entities. It also enables participants in a collaboration, and observers, to evaluate the level of trust in the infrastructure's ability to enforce appropriate information flow constraints before going ahead with sharing data. In this chapter, we demonstrate that the approach proposed in this dissertation is well placed to serve as the foundations of a trust domain.

7.2 Mergers and Acquisitions Scenario

Mergers and acquisitions are one of the most common transactions in the financial industry. According to Bloomberg [18], by the end of November 2011, a global total of 24,700 deals worth over $2.1 trillion were recorded, representing a volume change of 9% compared to the same period the previous year.

But what exactly are mergers and acquisitions? A merger is a transaction in which two corporations (normally in a similar or complementing line of business), of relatively equal size (i.e. in terms of market share) come together to create one large institution [11]. The assets of the two corporations are combined and the shares are redistributed. Acquisitions are similar to mergers in many respects, except that one of the corporations, referred to as the *target*, is relatively smaller compared to

[1]Trust Domains Project is a TSB funded project that aims to build a framework for enabling controlled information sharing.

the purchasing company, referred to as the *acquirer*. M&As are commonly carried out to improve efficiency, market share, cost saving through staff reduction, improve purchasing power and increase competitiveness.

To demonstrate how trust domains could be utilised to provide secure information sharing, we consider a scenario, illustrated in Figure 7.1, in which members of an M&A

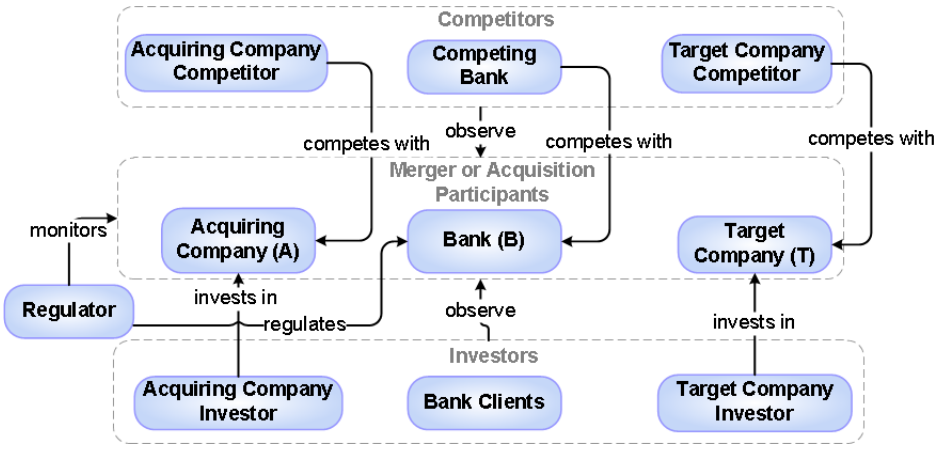


Figure 7.1: An M&A involving the take over of a company, T, by company A

team at a bank, B, use information, such as financial records and intellectual property, from the target and acquirer to perform various kinds of analyses including target company valuation, risk analysis and profitability analysis. However, such information is so sensitive that if leaked to unintended parties such as traders could have far reaching consequences on the success of the transaction as well as on the future of the parties involved. For this reason, both the parties involved and regulators, such as the Financial Services Authority[2], have a vested interest to ensure that the environment in which an M&A is performed would adequately enforce appropriate information flow constraints. To achieve this, a trust domain could be used to enable secure information sharing among participants in an M&A transaction. So how can the approach proposed in this dissertation help?

In the rest of this chapter, we answer this question by demonstrating how the approach: i) fits into the design of a trust domain-enabling architecture; and ii) can be used to provide evidence and mechanisms that use this evidence to enable evaluation of trust-properties for a trust domain.

[2]The Financial Services Authority was a UK body tasked with regulating financial transactions.

7.2.1 High-level View of an Infrastructure for M&A

We discussed, in the previous section, the need for participants to share information in an M&A. In this section, we describe a high-level view of an infrastructure, illustrated in Figure 7.2, that can be used to support this requirement.

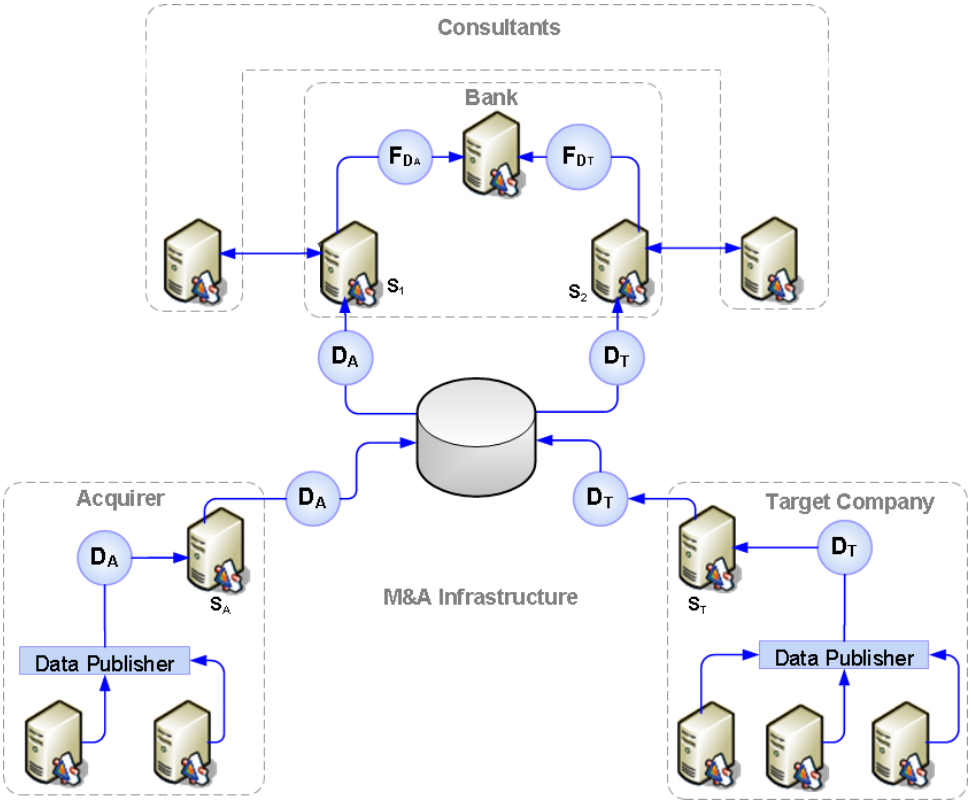


Figure 7.2: High-level view of an infrastructure for M&A

The target company sets up a system, S_T, with an appropriate software stack and policies and passes the data D_T to it. The data is then stored in a central storage location such as a network file-system. Similarly, the acquirer sets up a system, S_A, and passes the data D_A. The bank also sets up several systems that will be used by the M&A team. These systems access the data D_A and D_T for use in various kinds of analyses. Such analyses may be outsourced to external consultants, in which case the consultants set up systems that interact with the systems from the bank to access data relevant for the particular analyses being carried out. The analyses produce results (i.e. new data FD_T and FD_A), which is shared with bank executives to enable them make decisions about the particular M&A transaction.

7.2.2 Use Cases: The Need for Trust Domains

In this section, we consider two use cases that motivate the need for a trust domain-enabling architecture, discussed in Section 7.3.

Use Case I: Configuration Management of Services

The environment for an M&A scenario will comprise of a number of systems whose state will change as software is updated or new software applications are introduced. These changes can be managed through a configuration management process — an essential process for any large information system [72]. This process would ensure that systems that change into unacceptable configurations are identified and prevented from accessing the data and that procedures are in place to change these configurations into acceptable ones.

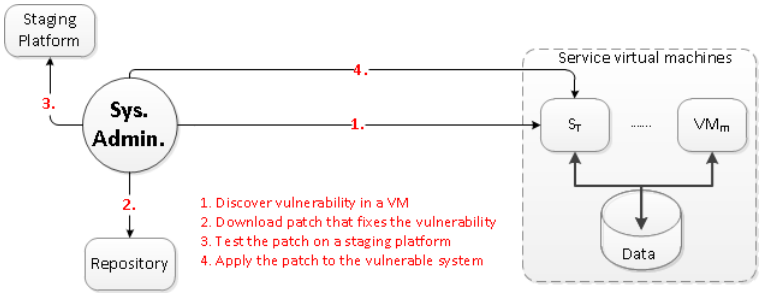


Figure 7.3: A use case involving configuration management on a system, S_T

We consider a use case, illustrated in Figure 7.3, in which a system administrator discovers a vulnerability in one of the applications in the software stack of S_T (step 1). At this point, the system can be considered to be running under some configuration, *RuntimeServiceSpecs* (RSS), which may differ from some initially applied configurations, *AppliedServiceSpec*. This is because vulnerabilities can be discovered and exploited at any point after applying the initial configuration. When the system administrator discovers that a patch is available to fix this vulnerability (step 2), they download the patch. However, before applying the patch, the system administrator needs to make sure that the patch will not affect other aspects of the system. In step 3, the patch is tested on a staging platform before being applied on S_T.

Use Case II: Trust Domain Compliance Checking

M&As are closely regulated by government agencies and independent bodies to ensure that their impact on local and global economies is controlled. This is achieved by

evaluating the environment in which an M&A takes place before, during and after the deal is complete. We describe a use case, illustrated in Figure 7.4, in which a regulator seeks to establish that the environment in which an M&A deal is performed conforms to the required compliance policies.

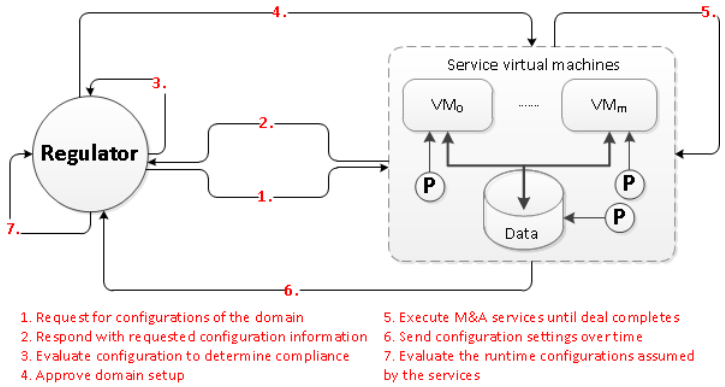


Figure 7.4: A use case involving a regulator performing compliance checks

In step 1 (not shown in the figure), the bank submits a description of the systems that will be used for the trust domain together with services they will offer to a regulator. This information describes the intended environment for an M&A and is used by the regulator to determine whether or not such an environment will conform to expected policies. We refer to this as *ServiceSpecs*.

Once the environment for the M&A has been set-up, the regulator requests for configuration settings of the infrastructure (step 2). This will include information about configurations of the systems, *AppliedConfigurationSpec* and services, *AppliedServiceSpec*. In step 3, the domain responds with the requested information which is evaluated against compliance policies in step 4. If evaluation results indicate compliance, the regulator gives a go ahead for the domain, allowing its execution, in step 5. During this period, the domain sends periodic updates, in step 6, about the runtime configurations of the services and systems. We refer to these as *RuntimeServiceSpecs*. In step 7, the collected configurations are evaluated to determine whether or not the domain's operation is or was in accordance with compliance policies.

7.2.3 Requirements

A trust domain must provide mechanisms to evaluate: intended configurations, i.e. *ServiceSpecs*; configurations applied on its constituents, i.e. *AppliedServiceSpec*; and configurations that its constituents are operating under, i.e. *RuntimeServiceSpecs*. To achieve this, a trust domain aims to satisfy the following requirements:

1. `TDOM-REQ 1: secure data storage` — the system must ensure that any data used or produced on any system that is used by the participants can not be compromised, even if the system is stolen or compromised.

2. `TDOM-REQ 2: secure communication` — any information exchanged among participants in an M&A must not be accessible to non-participants.

3. `TDOM-REQ 3: support custom services` — each participant in an M&A may require to perform analysis using their proprietary tools. The system should therefore support deployment of custom services for use in an M&A.

4. `TDOM-REQ 4: evidence collection` — the system must collect evidence that supports the state and/or characteristics of entities within the trust domain.

5. `TDOM-REQ 5: explicit composition` — the system must provide mechanisms for determining the constituents of the system, their states and how these constituents work together.

6. `TDOM-REQ 6: trust evaluation` — the system must provide mechanisms for evaluating the level of protection and data flow control that it provides.

7. `TDOM-REQ 7: access control` — define mechanisms for controlling membership and access to the data among participants.

The approach proposed in this dissertation can help towards achieving requirements `TDOM-REQ 4`, `TDOM-REQ 5` and `TDOM-REQ 6`.

7.3 A Trust Domains-enabling Architecture

In this section, we describe an architecture that can be used to provide trust domains for supporting an M&A scenario. We describe the components in the architecture before discussing how they fit together.

7.3.1 Components

An architecture for supporting trust domain sould consist of multiple components. In this section, however, we concentrate on those components that help with meeting requirements `TDOM-REQ 4`, `TDOM-REQ 5` and `TDOM-REQ 6`. These include a configuration manager, evidence aggregator, trust evaluator, policy manager, domain manager, build system and trusted platforms. Other components not discussed here include authentication and authorisation services, a storage service and a key manager.

Configuration Manager

Since a trust domain will consist of systems that may reside across multiple geographic locations and operated by different administrative organisations, one aspect that will require attention is the management of systems within a trust domain. More specifically, a trust domain deals with changes to the systems including those triggered by updates to the existing software/hardware components or installation of new ones and policy changes. Huh et al. [72] identified that configuration management will be a critical aspect of any distributed system that involve multiple administrative domains. Each administrative domain would need to maintain a list of configurations that are considered acceptable and keep this list up to date. Furthermore, any changes to this list are communicated to other administrative domains that are part of a trust domain. This is achieved by a *configuration manager* component, *CM*.

Evidence Aggregator

As services are being configured, policies being enforced or data being used in a trust domain, information about the activities performed by systems within a trust domain is collected during the lifespan of a trust domain. This information is collected in a systematic way to enable enable immediate or future trust evaluation. This is provided by an *evidence aggregator* component. This component interacts with evidence collection mechanisms such as monitors, provenance systems and integrity measurement systems that run on each of the systems that are part of a trust domain. The evidence aggregator manages the storage and retrieval of such information when requested by other components.

Trust Evaluator

In the use case on compliance checking, we discussed the need for a trust domain to provide mechanisms for evaluating the level of trust in a trust domain before, during and after its operation. In the previous section, we have discussed that evidence to support trust evaluation will be collected and stored to enable other components to use it. The *trust evaluator* is the main component that uses this evidence. This component combines various bits of evidence in a manner that enables evaluation of trust-properties e.g. by regulators.

Trusted Platforms

The evidence discussed above is used in trust evaluation. In order for such evaluations to be sound, this evidence must have integrity. This means that any component that comes into contact with such evidence, including the collecting, storage and reporting components, cannot modify the evidence without such modifications being detected. We require that each system that is part of the trust domain be a *trusted platform*, which supports integrity measurement, secure storage and attestation and provide roots of trust upon which trust in the evidence reported from it can be built.

Domain Manager

A trust domain has a lifespan through which its services are available and after which its operation may not be considered trustworthy. This lifespan is managed by a *domain manager* component. The domain manager handles requests for joining or leaving the trust domain and evaluates the configurations of systems against domain policies before admitting them into a trust domain.

7.3.2 Fitting the Components Together

Figure 7.5 illustrates the components and how they fit together into an architecture for enabling trust domains. A system administrator from a participating institution specifies the configurations of the kind of system that should be set-up to: i) enforce appropriate policies on the data; and ii) to provide services to other participants in a trust domain. These configurations include operating system and application requirements as well as network and storage settings. These are received by the CM, which then sends a request to the *build manager* to build a system according to these configurations. The build manager prepares the system by downloading and installing the relevant components from software repositories. Once completed, the resulting system is sent to the domain manager and the CM is notified.

The CM sends a request to the domain manager to admit the configured system into the trust domain. In addition, the CM specifies any configurations that should be applied to the system at runtime. For example, the CM could specify whether or not the system can be moved to another trust domain or how often the system should run updates on the underlying software. The domain manager uses the policy manager to determine whether or not the system satisfies the domain policies before admitting it into the trust domain. If the system fails to satisfy domain policies, a

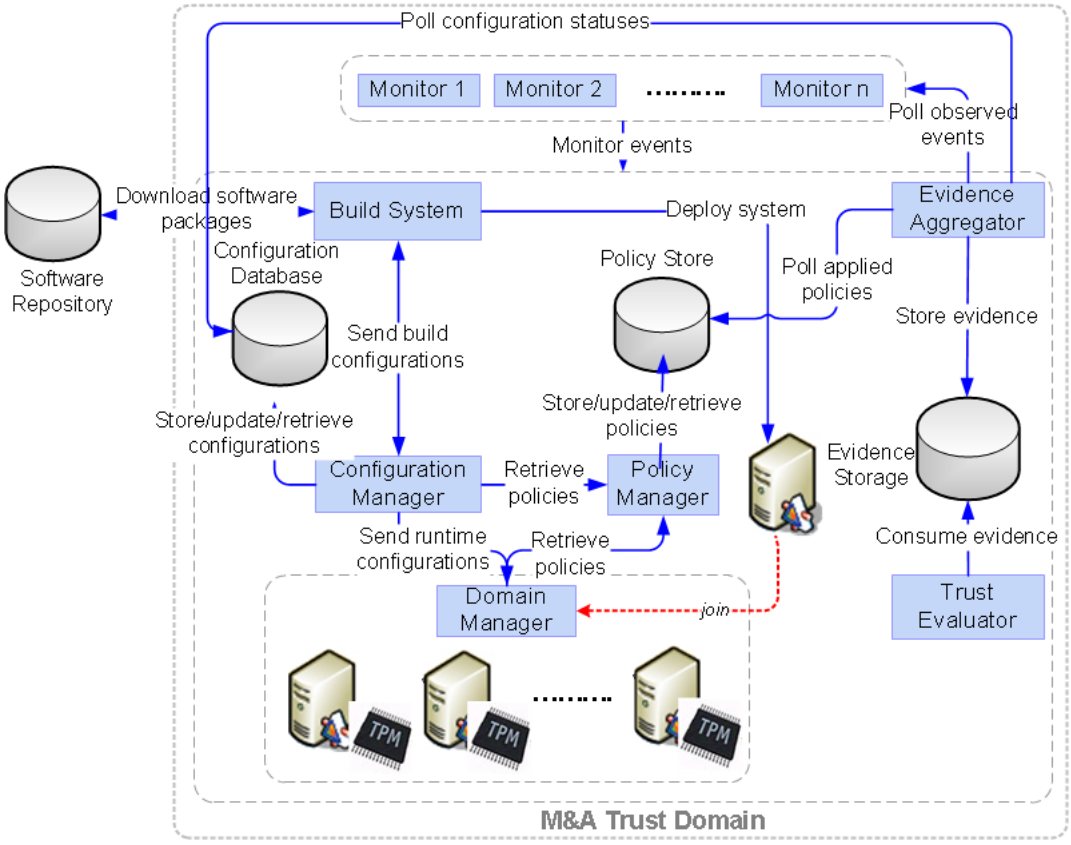


Figure 7.5: An architecture to support trust domains for M&A

message is sent to the CM indicating the configuration changes required. The CM propagates this message to the requestors, system administrators.

A number of monitors run during the lifespan of a trust domain. Each monitor observes events of other entities in a trust domain and maintains a record in some storage mechanism such as database or HDFS [157]. This information is used by the evidence aggregator, which uses a polling mechanisms to extract data about the events, configurations and activities of elements of a trust domain. The evidence aggregator reformats this information where necessary and provides an interface through which this information can be accessed. The trust evaluator uses this information and provides an interface that can be used by participants and regulators to query the state of a trust domain or a particular subset of it. The trust evaluator supports: i) standard queries that are based on compliance policies; ii) ad-hoc queries based on property templates; and iii) data export for analysis using other tools.

Once the parties involved agree to end the transaction, a request is sent to the domain manager to destroy systems within the trust domain. The domain manager, with the help of the policy manager, removes all the sensitive data, such as crypto-

graphic keys, financial analysis information, and destroys the entire filesystems used by these systems.

7.3.3 Design Evaluation Process

One of the key requirements for a trust domain, `TDOM-REQ 6`, is the ability to enable would-be participants or regulators to evaluate the level of trust in a trust domain. The architecture presented in the previous section enables this by providing mechanisms for collecting evidence, formatting it in a manner suitable for trust evaluation and using it in various kinds of reasoning. However, there are some other processes that occur outside of the architecture which may influence the properties of a trust domain. For example, the choice of component implementations included or the level of certification (e.g. Common Criteria [78]) could affect the overall properties of a trust domain or increase the confidence that relying entities have in a trust domain. For this reason, it is important that a design evaluation process — in which the components selected to be part of a trust domain are characterised and their combination is evaluated to determine whether or not the composition as a whole exhibits some desired properties — is included as part of a trust domain set-up. This process is achieved by dividing the lifecycle of a trust domain into three phases — *initialisation*, *operation* and *destruction* — collecting information or evidence in each phase and complementing this evidence with that obtained from the architecture.

The initialisation phase involves identifying and selecting components that will be included in the trust domain architecture, specifying them as well as providing initialisation data necessary for these systems to operate. Several kinds of evidence can be collected in this phase, including design documentation such as component specifications, compilation scripts and data [97] and component evaluation such as Common Criteria [78]. This evidence is especially useful for creating models presented in Chapters 4 and 5.

The operation phase involves taking the initialised systems, launching them and adding them to a trust domain. In this phase the systems provide the services for which they are designed. The main kinds of evidence in this phase will include event logs and provenance — indicating which data items are created by what processes — integrity measurements and runtime component configurations. These are particularly useful for creating models for the runtime aspects as discussed in Chapter 6.

The destruction phase occurs after the end of the operation phase and once the trust domain has completed serving its purpose. This phase involves archiving the

system state and/or data, erasing the data and re-formatting the systems to make them unusable. Provenance and event logs are the main types of evidence in this phase.

7.4 Applying the Architectural Approach

In the previous section, we discussed the design of an architecture for enabling trust domains and how the approach can be incorporated into the architecture. In this section, we demonstrate how ideas proposed in earlier chapters can be applied to enable evaluation of trust-properties.

7.4.1 Identifying and Classifying Trust-properties

The first step in using our proposed approach involves identifying and classifying trust-properties. In the context of a trust domain, and that of the scenario above, such properties would enable stakeholders to establish that the environment that will be, is being or was used to serve the M&A transaction provides necessary protections to prevent leakage or misuse of data. Based on an analysis of the use cases discussed in Section 7.2.2, we identify some of the trust properties below and classify them as shown in Table 7.1, as follows:

- `TP1: timely release of information` — given some data, D_t^X, where t is the time at which the secrecy of data is no longer important to the participants in the set X, then D_t^X must not be available to any entity y not in X before time t. For example, even if information about future business plans of the target may be shared with the bank, it might be required to be kept confidential between the two until the deal has been agreed between the target and acquirer.

- `TP2: authenticity of facts` — given F_{D_i} as facts generated from data D_i, then upon receiving F_{D_i} an entity must be able to determine the link to D_i. For example, when consultants perform valuation of the assets of the target company, the bank must be able to relate the results of the evaluation to the information about the assets of the target company.

- `TP3: compliant resources` — given any resource, R_{D_i}, such that the resource can come into contact with data, D_i, which is meant to be kept confidential to the participants of the trust domain, then R_{D_i} must demonstrate its ability to protect this data from unauthorised access. For example, contracts

that specify the terms of the deal would need to be seen by participants of an M&A, but may be restricted from the public, and, therefore, all the resources that may come into contact with the contracts may be required to demonstrate how they prevent this data from leaking outside the trust domain.

- `TP4: data usage enforcement` — given some data D_i and a usage policy, P_{D_i}, then every entity that uses D_i, must have enforced P_{D_i}. For example, if the target company specifies that information about some IP should not be shared with the acquirer, then the bank would be expected to enforce this policy whenever they deal with such information.

- `TP5: resource configuration maintenance` — given a resource, R_{CM_i}, that is registered to a particular configuration manager, CM_i, CM_i will ensure that R_{CM_i}'s configurations are kept to those specified in the policy. For example, a policy may specify that systems that control access to data must have operating system updates applied whenever they are available would be enforced all systems that are registered with a particular CM.

Trust Property	Class	Justification
TP1	Outcome	Information release can be considered as an unwanted outcome resulting from activities in a trust domain
TP2	Shadow of the future	The facts exist only after the data has been processed and therefore represent some possible future state of the system
TP3	Competence	The resources employ mechanisms which, when put together, should ensure that appropriate protections are applied on the data
TP4	Shadow of the future	The application of the policy can be represented as some state of the system and, therefore, this property determines whether this was the state when the data was used
TP5	Composition	Resources need to have certain components that allow them to be registered and bound to a particular CM and the combination with the CM should allow these policies to be applied

Table 7.1: Assignment of the identified trust-properties to property classes.

7.4.2 Evidence and Architectural Models

In the architecture presented in Section 7.3, we identified evidence as being central to a trust domain. In this section, we demonstrate through some examples the kinds of evidence that can be collected and how they can be used to create architectural models.

Provenance as a Security Control

In Section 3.6.3, we specified that a runtime-focused approach is best suited for reasoning about properties that fall in the *shadow of the future* class, for which we have proposed the use of a provenance model in Section 6.4. We demonstrate here that such a model can be created to represent activities in a trust domain and cold serve as a security control.

The components identified in Section 7.3 could each comprise of executable programs, libraries and configuration files. Each component could also involve a number of processes in which these programs are executed, configuration files are used and libraries invoked. The components, or sub-components, might also communicate with each other to exchange information. These activities can be captured in a provenance model. For example, the process in which the build manager creates systems that are admitted into a trust domain could be captured in a model similar to that demonstrated in Section 6.4.3, where platforms in this case would imply systems admitted into a trust domain and a collaboration would be equivalent to an M&A transaction. Similarly, a model capable of enabling reasoning about `TP2` and `TP4` would include the processes in which data D_i and policy P_{D_i} are used and data F_{D_i} is generated.

In this case, provenance serves as a security control [104] — since provenance provides mechanisms for detecting unwanted changes to D_i and also serves as a deterrence towards participants who may be thinking about falsifying the relationship between D_i and F_{D_i}.

A Trust Domain as a Set of Co-operating Processes

In the previous section, we identified `TP1` as falling in the outcome-based class. As discussed in Section 3.6, we can reason about this class of properties by following an interaction-focused approach, for which a model and framework were presented in Chapter 5. We can use this approach by treating a *trust domain as a set of co-operating processes* that ensures that certain data flow patterns are not permitted, a necessity for `TP1`.

In this representation each constituent of a trust domain is represented by a process capable of performing one or more actions that transform data or transmit it across its interfaces. To verify trust-properties, these processes are composed together, using `Principles 1 and 2`, to create a process that can be checked to determine whether or not it is capable of creating interactions that permit unwanted data flows.

The model included in the framework can be instantiated with configurations that specify the computational logic of the components of a trust domain, data stored on the resources or interfaces available to other components. Using this approach, a trust domain can be instantiated as shown below:

```
Domain_Manager_Logic(id) = □ c: PotentialCMCode •
        req_deploy.c.id.rtmcfg → download.id.R.getVALocation(rtmcfg) →
        deploy.id.getVA(rtmcfg) → join.id.getVA(rtmcfg) → ...

Configuration_Manager_Logic(id) =  receive.id.SysAdmin.cfg →
        □ b: PotentialBuildManagerCode •
                req_build.id.b.cfg → res_build.b.id.Msg →
                □ d: PotentialDomainManagerCode •
                        req_deploy.id.d.rtmcfg →....

Evidence_Aggregator_Logic(id) = □ m: PotentialMonitorCode •
        pool.id.m.e → store.id.S.e → ...
        {*where S is the identity of the storage system *}
....

Tdoms_ResourceCfg = (M, Tdoms_Memory_Contents)

Tdom_Cfg = <(dm, l1), (cm, l2), (ea, l3)...>
        {*l1, l2, l3, ..are locations in memory *}

Tdoms_AppCfg = <(dm_code, dm, Domain_Manager_Logic, {join, deploy,...}),
        (cm_code, cm, Configuration_Manager_Logic, {receive, req_deploy,...}),
        (ea_code, ea, Evidence_Aggregator_Logic, {pool, store,...}),.... >
        {*cm, ea and dm are roles corresponding to each component and
         cm_code, cm_code and cm_code are corresponding code identities *}

M = System(Tdom_Cfg, Tdoms_ResourceCfg, Tdoms_AppCfg)
```

Systematic Characterisation of Trust Domains Building Blocks

A trust domain comprises of several components and systems that may be trusted for different aspects. For example, the monitors may be trusted to securely collect evidence but not to securely store it. For this reason, various kinds of protection mechanisms may be applied on each system to ensure that it operates as expected. Some of these mechanisms were identified and characterised in Chapter 4. These can be used as building blocks for trust domains. For example, a model could be instantiated as shown below.

$$DomainManager, ConfigurationManager : Component$$
$$TrustEvaluator, EvidenceAggregator, PolicyManager : Component$$
$$cm, dm, pm, ea, te : NAME$$

$$
\begin{array}{|l}
tdom : TrustworthySystem \\
\hline
tdom.component = \{dm \mapsto DomainManager, cm \mapsto ConfigurationManager, \\
\quad ea \mapsto EvidenceAggregator, pm \mapsto PolicyManager, te \mapsto TrustEvaluator\} \\
tdom.assigned = \{DomainManager \mapsto \{a1\}, ...\} \\
tdom.access = \{DomainManager \mapsto \{a2\}, ...\} \\
tdom.privilege = \{DomainManager \mapsto 3, ConfigurationManager \mapsto 3, ...\} \\
tdom.TCB = \{EvidenceAggregator, ConfigurationManager, ...\}....
\end{array}
$$

7.4.3 Property Verification

The last step in applying the architecture-based approach is verification of trust-properties. This step involves specifying the properties in a manner suitable for verification against a specific model and verifying them against that model. We have already classified the properties and developed suitable models for capturing aspects that can be used to reason about properties in each class. In the rest of this section, we specify the properties in a suitable formalism and demonstrate how these can be verified against the models developed in the previous section.

Verifying TP1

Since `TP1` falls in the outcome-based class, which can be reasoned about using the abstraction framework presented in Chapter 5, it can be specified using CSP, as shown below. Here X, D and t represent the constituents defined for D_t^X, above, and Y is a superset of X.

```
TP1 = TP1'(X, D, t)

TP1'(X', D', t') =
        if t '> now
                □ x,y:X' • send.x.y.D' → TP1(X',D',t')
        else
                □ x,y:Y • send.x.y.D' → TP1(X',D',t')
```

This property is verified by creating an abstraction of the instance, M, such that only events that occur over the *send* channel are included and then performing refinement against the resulting abstract process, as shown below.

```
AbsSystem = M\diff(Σ, {send})
assert TP1 ⊑_T AbsSystem
{*Where M is the system instantiated as shown in the previous section *}
```

This would result in either the failure or success of the assertion. Failure occurs when the constituents are configured to allow interactions over the *send* channel in which D is passed between an entity in the set X and one not in X.

Verifying TP2 and TP4

TP2 and TP4 seek to determine whether or not the relationship between two pieces of data, D_i or F_{D_i} and D_i or P_{D_i}, respectively, is appropriate. This data will exist when the trust domain is in operation and therefore fall in the *shadow of the future class*.

In the previous chapter, we have proposed two approaches for reasoning about these properties and, therefore, two options of specifying and verifying the properties. While in certain cases, the two approaches can be used in tandem, in other cases, one of them may suffice. In the case of the two properties, TP2 and TP4, a provenance-based model is sufficient. Listings 7.1 and 7.2 shows how queries for the two properties can be formulated. For TP2 the integrity of the data used and generated from processes on a system together with that of the programs executed is returned, and for TP4 the query returns the processes that used some data of interest, D_i in this case with some integrity value "xy1235i", together with the policy that was applied.

```
SELECT { ?d ?x ?fd ?y ?z ?e}
WHERE {
        ?p rdf:type :Process .
        ?p :used ?d .
        ?p :wasPerformedBy ?e .
        ?fd :wasGeneratedBy ?p .
        ?x :integrityOf ?d .
        ?z :integrityOf ?e .
        ?y :integrityOf ?fd . }
```

Listing 7.1: Example query to capture TP2

```
SELECT { ?d ?p ?pd ?y}
WHERE { ?x :integrityOf ?d .
        ?y :integrityOf ?pd .
        ?p :used ?d .
        ?p :used ?pd .
        FILTER ( ?x = "xy1235i"^^xsd:string )}
```

Listing 7.2: Example query to capture TP4

Verifying TP3 and TP5

TP3 and TP5 are both concerned with identifying mechanisms, and their capabilities, employed on the systems that are part of a trust domain. These properties fall in the competence and composition classes, respectively, and can therefore be verified using structural aspects of a system. In Section 4.4 we discussed that trust-properties are specified using predicate logic — where each property is captured as a predicate formula. Using this approach TP3 can be specified as a predicate that checks whether a given instance is of a certain type, where the type is determined by the definition of a complaint resource. For example, if compliance implies being based on trustworthy architecture, then a predicate such as the following could be used.

$\forall\, r : \mathit{Resource} \mid r \in \mathit{contact}\ d \Leftrightarrow r \in \mathit{TrustworthySystem}$
where d is some data, Resource is a set of systems and
contact is a relation which specifies systems have access to specified data

TP5 could be specified in a similar manner, but would concentrate on determining the presence of a particular elements on the resource.

7.5 Evaluation

In the previous sections, we have demonstrated how the approach presented in this dissertation could be used as foundations for a trust domain. In this section, we evaluate the benefits and challenges of such an approach and how it compares with alternative approaches.

7.5.1 Benefits for Trust Domains

The application of the proposed approach could benefit trust domains in several ways, some of which include the following.

Explicit Basis for Trust

One feature of trust domains that distinguishes it from other similar concepts such as virtual organisation (VO) [46] or trusted virtual domains [61] is the ability to be specific about where trust is being placed. Rather than simply allow parties to identify each other, as is the case with VOs, each party can determine the level of trust in another or in the trust domain as a whole. This is enabled by the idea of making the state and characteristics of a trust domain explicit through the use of an approach such as the one proposed in this dissertation. Evidence is collected from each participant or system in a trust domain and used to determine its current state, and possible future states. This allows each trust decision to be linked to the properties that can be determined about the state of the entities in a trust domain and thus making the basis of trust explicit.

Multiple Perspectives

A trust domain is used by multiple stakeholders each interested in one or more aspects of a trust domain. For example, in the M&A scenario regulators are interested in the ability of a trust domain to prevent information leakage whereas the M&A team from the bank might be interested in the accuracy of analyses performed in the trust domain. In order for these stakeholders to reason about things that they care about, mechanisms must be put in place to collect evidence that may be useful for a given perspective. The approach presented in this dissertation provides good abstractions

that can be used to serve different perspectives, and thus, allowing stakeholders to evaluate the properties of interest, from their own perspective, before making use of or joining a trust domain.

Inputs for Economic Models

The Trust Domains project aims to enable evaluation of economic aspects of setting up a trust domain versus the security benefits. The idea is to simulate each possible configuration of a trust domain against some economic indicators to determine the best way of setting up a trust domain for a given purpose. This requires a characterisation of the building blocks of a trust domain and evidence that they can produce when a trust domain is enacted. The approach presented in this dissertation provides such a characterisation; the semantic model provides a means of capturing the characteristics of the building blocks in a manner that enables one or more aspects of the trust domain.

Dynamic Trust Evaluation

The approach proposed in this dissertation enables collection of evidence at various stages of a trust domain. For example, regulators may want to evaluate the level of trust before a trust domain is setup or after the M&A transaction is complete. This enables dynamic evaluation of trust properties in that evidence can be initially collected from the possible constituents of a trust domain and continues to be collected even when a trust domain is in operation. This evidence is made available for trust evaluation before, during and after the operation of a trust domain.

7.5.2 Practical Considerations

The approach proposed in this dissertation can be employed during the architecture design and analysis phase of a system. An architect would begin by identifying the properties they would like the system they are designing for to satisfy. System design begins with a set of functional and non-functional requirements. These requirements are a good source from which to identify some properties. Properties would either be some constraints on the requirements or some desirable attribute that supports the requirements.

With the properties identified, an architect would use the classification presented in Section 3.4 to classify the properties and identify the kind of evidence that would be helpful for each class of properties. The next step would be to obtain the identified

evidence. This evidence would normally come from design documentation of the system or similar systems. Runtime evidence can also be collected if the system is operational or from system prototypes.

The collected evidence is then modelled using one, or a combination, of the proposed models. In practice, this evidence will not entirely be in line with the requirements of the model and, therefore, an architect would be required to use their skills and judgement to decide which pieces of evidence to use and which to leave out. The important point here is that the architect would be *explicit* about their choices, enabling team members and stakeholders to have a say.

Once the system has been modelled, the architect can then perform property verification. Currently, the majority of the work would be done using tools such as *FDR*, *fUZZ* [160] and *Prob*, while some aspects may still need to be performed manually. However, our experiences have shown that one can quickly run into state space explosions. One way of avoiding this problem is to have a good choice of abstractions. That is, one would have to look at the evidence again and decide if there is still anything that can be ignored without losing essential information.

Future work might involve integrating the approach into well known design frameworks as well as providing better tool support.

7.5.3 Challenges

In this section, we discuss some of the challenges faced by the proposed approach.

Multiple Formalisms

The approach presented in this dissertation makes use of multiple formalisms to demonstrate various aspects. Each formalism is particularly good for representing a particular aspect. For example, the use of a CSP-based model enables modelling of concurrent executions, a necessity when reasoning about interactions. While this approach demonstrates that such formalisms can complement each other, it might provide a challenge to users since they may be required to understand each formalism to make better use of the approach.

Evolving Trust Domains

The concept of a trust domain is still under development. While the work presented in this dissertation has the opportunity to shape the direction of trust domains, there is a risk that it might require substantial changes as the notion of a trust domain evolves.

For example, the current inclination has been to enable trust evaluation through provision of evidence of state and behaviour of entities within a trust domain, which fits in well with our thesis. However, if the focus is shifted to providing techniques for enabling secure information sharing without demonstrating how secure this makes a trust domain, then our approach would need some changes to make it more suitable.

7.5.4 Related Work

Having realised the complexity of correctly configuring virtual infrastructures to conform to enterprise security policies, Bleikertz et al. [17] have described an automated virtual infrastructure validation and assurance system. Their system unifies the configurations of virtualisation infrastructures (e.g. virtual machines) into a generic data model. This data model is then visualised through a data flow graph, which in turn, is used to detect any breach of security requirements.

Similar concepts to a trust domain have been applied in other contexts to control information flow. Griffin et al. [61] proposed the concept of a trusted virtual domain (TVD) which defines a group of virtual machines that are configured to enforce a common policy. However, this concept does not provide mechanisms for evaluating the properties of a particular TVD. The approach proposed in this dissertation could be applied to TVDs so that both the policy applied and activities of the TVD members can be evaluated.

Virtual organisations [46] and grid trust domains [48] have been proposed as means of controlling information and resource sharing in grid computing. Each VO or grid trust domain consists of organisations or individuals and resources that they share. Membership is based on credentials such as certificates and not mechanisms are provided to determine or enforce the behaviour of members. Similar to TVDs, VOs can benefit from the architectural proposed in this dissertation to characterise systems and resource within a VO.

7.6 Conclusions

The notion of a trust domain has been proposed as a means of providing mechanisms for enabling secure information sharing based on the state or characteristics of entities involved in a collaboration. It enables participants in a collaboration, such as a mergers and acquisitions transaction, to specify constraints on information flow while providing mechanisms for evaluating the level of trust that can be assigned to it. To

achieve this, trust domains require evidence to be collected from their constituents and used to reason about trust properties.

In this chapter, we have demonstrated that the approach proposed in this dissertation can form the foundations for trust domains. Using this approach, evidence collected from the constituents of a trust domain can be aggregated and presented in a manner suitable for evaluation of properties using various aspects of a trust domain.

Chapter 8

Conclusions and Future Work

In this chapter, we summarise the contributions and evaluate the strengths and weaknesses of the proposed approach before giving a conclusion and discussing possible future work.

8.1 Contributions

The need for computer systems to operate in a trustworthy manner and to be able to demonstrate this to those relying on them has never been greater. Towards this end, trustworthy system architecture design focuses on including components in a generic system architecture that enhances the system's ability to record and report its configurations and to securely store sensitive information such as cryptographic keys. However, such systems work on the assumption that by identifying the components on a system then one can be assured about the behaviour of the system. State-of-the-art focuses on the use of trust-enabling mechanisms to support trustworthy behaviour in specific application domains. For example, trustworthy grid architectures proposed by Cooper [31] and Löhr *et. al.* [94] propose the use of remote attestation in combination with virtualisation to enable a trustworthy execution environment of grid jobs.

While the use of trust-enabling mechanisms serve a useful purpose in system design, the lack of a systematic characterisation of these mechanisms prevents evaluation of the influence of these mechanisms on trust-related properties. Consequently, the basis of trust in such systems and the assumptions upon which it is based are not made explicit. This leads to the creation of systems that use related concepts but cannot themselves be related in a meaningful way.

In this dissertation, we have proposed an architectural approach as a means of establishing the properties and assumptions upon which trust in a system is based. This approach involves identifying classes of trust-properties and mapping each class

to some aspect of a system that can be used to reason about them. This mapping motivates a number of requirements for models that can be used to represent knowledge about the system from a given perspective. We build these models and demonstrate how they can be used to reason about trust-properties. Each model is formally defined and a framework that guides the use of the model is developed.

A model of trustworthy system is developed using Z to capture the characterisation of concepts in a system architecture and trust-enabling concepts, which are then combined to create a model of a trustworthy system. This makes the properties and assumptions of the elements and how they contribute to the overall properties of the system explicit. We build an abstraction framework based on CSP to model how interactions can be used to reason about *outcomes* from a system. This framework enables systems with varying properties to be constructed using different configurations and to reason about concurrent execution of components within a system architecture. We develop trace- and provenance-based approaches to capture runtime aspects of a system. In these approaches, evidence about the activities of a system is collected and represented in a form that enables evaluation against properties that seek to relate the past, current and possible future behaviour of a system.

8.2 Evaluation

In this section we evaluate the strengths and weaknesses of the approach proposed in this dissertation.

8.2.1 Managing the Size of Evidence

One possible bottleneck with this approach is the amount of evidence that is generated. In Section 7.4.2, we identified several kinds of evidence that can be collected during the phases of a trust domain. Some of this evidence needs to be continuously collected. For example, evidence during the operation phase of the system such as events logs need to be collected throughout the operation of a trust domain. As a result, the size of evidence will continuously increase. This size may lead to a number of challenges including: i) maintaining its integrity; ii) computational effort required to process it; iii) noise ratio; iv) storage requirements; and most importantly, v) ability of relying entities to make sense out of the evidence.

Several large scale distributed data storage systems such as Hadoop [175] exist and could provide storage and computation capacities at a much larger scale than the trust domain discussed in this chapter. These systems also provide confidentiality,

through encrypted storage, and integrity, through cryptographic hashing and multiple backups. In fact a sub-project of Hadoop, Chukwa [137], could be well suited for this.

The issue of noise ratio is one that is common to any data collection task. It is particularly challenging for our approach to determine what data is noise and which is not. This is because the use of architectural aspects imply that while one piece of data may be considered noise in one aspect, it might be a key data item for another aspect. The use of the provenance-based model proposed in Section 6.4 could help towards addressing this issue. Semantically rich provenance information could help distinguish noise from good information [104]. Furthermore, such semantics, especially those based on a graph as discussed in Section 6.4.1, enable this information to be split into smaller chunks, such as sub-graphs, which only include essential data, and thus isolating any noise from useful data.

8.2.2 Trusting the Collection Mechanisms

One of the main assumptions placed on the approach is the availability and integrity of evidence used in creating the models and, consequently, on the availability of trustworthy collection mechanisms. Indeed continued work on provenance, trusted logging and virtual machine introspection indicate that substantial research and development need to be conducted to achieve reliable evidence.

Adopting an architectural approach also implies a reliance on documentation about the functions, interfaces and capabilities of components used in a system. Indeed, several functioning systems lack clear architectural descriptions while the adoption of agile methodologies imply that less documentation will exist before a system is built. However, this problem can be eased with the runtime approach by setting up the system in a test environment and discovering architectural aspects using mechanisms such as runtime architecture discovery [7, 181, 69].

8.2.3 Impact on Future Designs

While the work presented in this dissertation has focused on existing systems, it is important to put this into the perspective of the future. Therefore, the natural question that arises is that of how can the approach influence the design of trustworthy systems?

First we discussed in Section 2.3 that while there is continued increase in the use of trusted computing techniques to enable trustworthy system operation, the basis for the trust in resulting systems is not clearly understand. This is born from a lack of a

systematic characterisation of the building blocks and relationships among them. By proposing to deal with composition and competence separately from interactions created or processes performed on a system, we can begin to look into more detail about the nature of the building blocks and how their combination results in trustworthy behaviour.

Second, the idea of using an architectural approach not only helps to manage complexity but also we can begin to build multiple interfaces into trustworthy systems

8.2.4 Privacy Concerns

Applying such an approach to trust domains implies collecting as much information as possible about the activities of the entities within a trust domain, as specified in `REQ 10`. This could imply that activities about personal transactions could also be included. In such cases increased use of provenance might lead to increased security and privacy risks [104]. Furthermore, the idea of identifying and explicitly stating relationships among the entities and data in a trust domain implies that several deductions can be made beyond those required for verifying properties.

This problem is, however, not limited to our approach. Indeed, any system that requires monitoring or collection information about activities performed on a system used by some users has the potential to breach the privacy of those users. To overcome this problem, a number of approaches could be followed. A risk-based approach could enable would-be participants to weigh the potential impact of any privacy breach against the benefits of participating in a trust domain. Privacy-preserving mechanisms [4] could be employed to ensure that the properties can still be verified without compromising privacy. A trusted third party could also be tasked with the responsibility of managing such information.

8.3 Future Work

The work presented in this dissertation is only a part of the larger problem of establishing trust in computer systems. As a result, there is no shortage of possible improvements or extensions to this work. We discuss some of the interesting ideas that could be explored by future work.

8.3.1 Identifying Essential Configurations

The models presented in this dissertation consider some key data items as evidence that can be used to reason about trust. Indeed, the set of such evidence considered

here is not exhaustive. For example, though the provenance model generated during the build process of virtual machines includes essential evidence about reasoning about what could have transpired during such a process, it by no means restricts what sort of parameters a virtual machine monitor would pass to the VMs. There are possibly more configuration and other types of information that could be captured and used in reasoning. Possible future work may involve identifying this information and incorporating them into the model. This requirement is also echoed by Vishik et al. [173] who refer to this information as *trust evidence parameters*.

8.3.2 A Taxonomy for Trust Domains and Trust Indicators

On-going work on the Trust Domains project involves defining a taxonomy for trust domains and trust indicators. A taxonomy helps to capture concepts in a trust domain and relationships among them. It also defines inference rules which specify how new information can be learnt about the structure of a trust domain by using certain concepts. The work presented in this dissertation provides a foundation upon which such a taxonomy could be built. However, the approach does not include processes and procedures that may exist in a trust domain. This will be investigated as part of research work on the Trust Domains project.

8.3.3 A Common Semantics for Expressing Trust-properties

We have demonstrated how the use of an architectural approach helps to simplify the problem of evaluating trust-properties. Different models, with different semantics, are used to capture knowledge about the system from a given aspect. Though the models share some semantics, they require properties to be expressed using semantics used by a specific model. We envision that our work could benefit from using common semantics for expressing trust-properties. Trust properties could be captured in some abstract model that is translatable to the semantics used by each model.

8.4 Conclusion

The issue of reasoning about trust in a system is a particularly challenging one due to a number of reasons including: complexity of modern computer systems; the lack of a clear definition of trust in the context of a computer systems; and satisfying multiple parties that rely on the system. This problem renders itself well to an architectural approach. This is because an architectural approach helps to divide harder problems

into simpler ones that can be investigated and designed for separately and whose solutions can later be combined to solve the harder problem.

In this dissertation we have applied an architectural approach to provide a systematic way of establishing the basis of trust in trustworthy systems. We have demonstrated that by dividing a system into several aspects and categorising trust-properties, a mapping can be created between classes of trust-properties and aspects of a system that can be used to reason about such properties. However, this requires that knowledge is collected about each aspect and represented in a manner suitable for automated and formal reasoning. Towards this end, we have developed models and associated framework which capture evidence from a specific aspect and represented it in a model against which trust-properties can be verified. These models are an effective means of demonstrating how trust in a system develops from the properties of its constituents and how the combination of these elements induces system properties.

However, the approach depends on the ability to efficiently and correctly specify the properties. While this may be overcome by using some kind of knowledge sharing approaches such as that adopted in most patterns communities, it might be a problem for systems that are unique (i.e. they may not be enough information about such systems to take advantage of community effort in specifying the properties). Furthermore, the approach relies on existence of evidence and collection mechanisms that ensure that such evidence is protected against integrity and privacy breaches. This is an open problem which continues to receive considerable attention and it is our hope that flexible and scalable solutions will soon be available.

Appendix A

Schema of the Whitelist

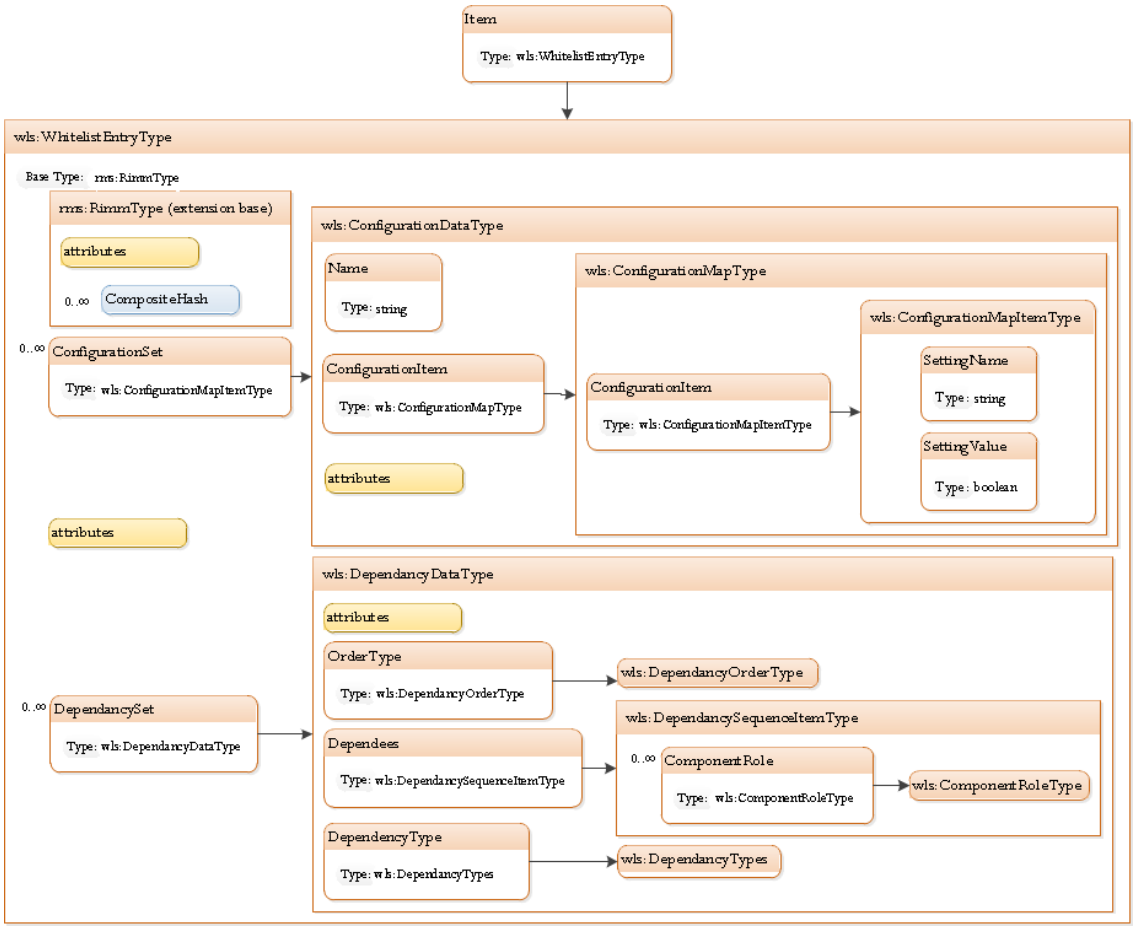


Figure A.1: Whitelist schema — an extension of *RimmType* from TCG RM Schema

A component's behaviour not only depends on its integrity but also on its configuration [59]. To capture this, we define a map, *ConfigurationDataType*, of all settings that may affect a component's behaviour to the values of those settings. For sim-

plicity, we use Boolean expressions for specifying these values, indicating whether a setting is activated or not.

In order to capture relationships among observed events in a system, we identify three types of dependencies, defined as *DependencyTypes* in the schema, that may exist between components in the system to include *Install*, *Execute* and *Config*. These capture the idea that a component may require another component to be installed, executed or configured in order for it to function as expected. Furthermore, a given dependency will be specified with *DependencyOrderType*, *Strict* or *Relaxed*, to indicate the constraints on the ordering of the dependencies.

We define *ComponentRoleType* to capture the roles in the system and *DependencyDataType* to represent the relationship of a component to other components that play given roles. *DependencyDataType* is captured as a sequence of roles (*Dependees*), the type of dependency (*DependenceTypes*), i.e. *Install*, *Config* or *Execute* and the order type (*Strict* or *Relaxed*).

We define *WhitelistEntryType* as a single entry on the whitelist, that extends the *RimmType* defined in the RM schema by adding a set of dependencies (*DependencySet*), a set of configuration settings (*ConfigurationSet*) and a unique identifier (*ID*). These entries are then combined into a set to complete the definition of the schema.

Appendix B

Z Primer

Z is a mathematical language based on set theory and first order predicate logic used for specifying and designing software systems and reasoning about the properties of such systems. The design is captured using standard set theory operators such as set comprehension, cartesian product and power sets [178] and a schema language provides mechanisms for structuring and composing specifications.

In this dissertation we make signifant use of the schema language. Therefore, in this chapter, we discuss the notation and semantics of the main Z concepts used in this dissertation. Refer to Spivey [159], Jacky [81] and Woodcook and Davies [178] for a comprehensive discussion of the set operators and the underlying mathematical theory.

In Z schemas are used for structuring and composing descriptions. They are used in three main ways: as a description of new types; as predicates or as declarations. We give examples of each approach in the following sections. We also discuss the important schema operators and how system operations are modelled using the schema language.

B.1 Schemas as Data Types

When used as descriptions, schemas comprise of state variables and constraints on them. The constraints serve as invariants, so that any valid state of a given schema will satisfy the invariant. For example, given a schema A, then P is considered an invariant on the state of A and any valid state of A will satisfy P.

$$
\begin{array}{|l}
\hline
A \\
x : X \\
y : Y \\
\hline
P \\
\hline
\end{array}
$$

B.2 Schemas as Predicates

Schemas can also appear in predicates. For example, the schema

$$\forall\, x : X;\ y : Y;\ w : W \bullet A$$

has the effect of introducing the predicate P defined in the schema A, as follows:

$$\forall\, x : X;\ y : Y;\ w : W \bullet P$$

However, there is a requirement that the declaration part must include all the state variables included in the declaration part of schema A ($x : X$ and $y : Y$, in this case).

B.3 Schemas as Declarations

Schemas can also be used as declarations. For example the axiom

$$\exists\, A \bullet \ldots$$

has the effect of introducing all the state variables declared in the schema A. This is equivalent to the declaration:

$$\exists\, x : X;\ y : Y \bullet \ldots$$

B.4 Schema Operators

Schemas provide mechanisms for structuring specifications and provide mechanisms for composing simpler descriptions into more complex descriptions. Z defines a number of operators that can be used to compose schemas including schema inclusion, disjunction, conjuction and composition.

Given a schema A and B, where P and Q are some predicates, as shown below, then schemas C, D, and E represent the results of schema inclusion, disjunction and conjunction, respectively.

B

$x : X$
$w : W$

Q

The schema C describes the notation for specifying schema inclusion. In this notation, C includes schemas A and B. This has an effect that declarations of schemas A and B are combined such that any state variables defined in both A and B appear only once and the predicates are combined using logical $\wedge$, as shown on the right.

C

A
B

R

C

$x : X$
$y : Y$
$w : W$

$P \wedge Q \wedge R$

The notation for schema disjunction is shown on the left. The schema D is a disjunction of schemas A and B. In this case, declarations are combined in a similar way to schema inclusion and the predicates of A and B are combined using logical disjunction.

$D \mathrel{\widehat=} A \vee B$

D

$x : X$
$y : Y$
$w : W$

$P \vee Q$

Schema conjunction works in a silar way to disjunction, except that predicates are combined using logical conjunction. The schema on the right illustrates the effect of combining A and B using conjunction.

$E \mathrel{\widehat=} A \wedge B$

E

$x : X$
$y : Y$
$w : W$

$P \wedge Q$

Other operators not used in this dissertation include negation, composition and . The semantics of these can be found in any Z textbook such as Jacky [81] and Woodcook and Davies [178].

B.5 Operation Schemas

In this dissertation we make significant use of operation schemas. In Z, operation schemas are used to capture requirements and effects on state changes in a given system. Operators are defined using the same notations as state schemas except they include initial and after (called primed) states of one or more state schemas. For example, an operation, *ChangeX*, on the state schema A can be defined as shown below. In this case, the symbol ΔA specifies that the operation will change the state of A. Effectively, this has the effect of including both initial and final states of A. Final states are represented as primed, A' as shown on the right.

ChangeX

ΔA

$newX? : X$

$x' = newX?$

ChangeX

A

A'

$newX? : X$

$x' = newX?$

When operation schemas are used, it is important to indicate the effect of an operation on all state variables. For example, the operation *ChangeX′* is a deterministic version of *ChangeX*.

ChangeX′

A

A'

$newX? : X$

$x' = newX?$

$y' = y$

Operation schemas can also be combined using schema operators discussed above. We make significant use of this feature when defining the effects of performing more than one operation on a given schema and when promoting operations at the component level to the system level. For example, the operation *ChangeX*, defined above can be promoted to an operation on a system represented by schema F. F specifies

a system that has a function *setA* that maps a list of names to the components, A, of F.

To specify the effect of operation *ChangeX* at the system level, i.e. F, we first define a promotion schema, *PromoteX*, which relates the binding of A that will be affected by the *ChangeX* operation. The *PromoteX* operation is then combined using schema conjunction as shown below.

PromoteX

ΔA
ΔF
$nameOfA? : NAME$
$newX? : X$

$\theta A = setA\, nameOfA?$
$w' = w$

F

$setA : NAME \pfun A$

$ChangeXonC \mathrel{\widehat=} \exists A \bullet$
$PromoteX \land ChangeX$

The expression θA used in *PromoteX* refers to the particular schema binding specified in ΔA, i.e. the particular instance of A that will be affected by the *ChangeX* operation.

Appendix C

Complete Semantic Model

This chapter describes the complete semantic model.

C.1 Capturing the System Architecture Model

C.1.1 Data and Interfaces

Below are the basic data types that are used throughout the model.

$$Time == \mathbb{N}$$
$$Ring == 0 \,..\, 3$$
$$Root ::= Reporting \mid Measurement \mid Storage$$
$$[DATA, ADDRESS, KEY, NONCE]$$
$$[NAME, LOGIC, PROTOCOL]$$
$$SYSTEMSTATE ::= live \mid offline$$
$$INTERFACESTATE ::= enabled \mid disabled$$
$$COMPONENTSTATE ::= running \mid loaded \mid notloaded$$

Interface

$$intfState : INTERFACESTATE$$

InitInterface

$$Interface$$

$$intfState = disabled$$

ChangeInterfaceState

$$\Delta Interface$$
$$newIntfState? : INTERFACESTATE$$

$$intfState \neq newIntfState?$$
$$intfState' = newIntfState?$$

C.1.2 Components

Computation

$$triggerInterface : \mathbb{P}\, Interface$$
$$logic : LOGIC$$

Component

$$
\begin{array}{l}
opState : COMPONENTSTATE \\
execInterface, measureInterface : NAME \\
cmpInterface : NAME \pfun Interface \\
activeCMPInterface : \mathbb{P}\, NAME \\
cmpComputation : NAME \pfun Computation \\
\hline
measureInterface \neq execInterface \\
execInterface \in (\mathrm{dom}\ cmpInterface) \\
measureInterface \in (\mathrm{dom}\ cmpInterface) \\
activeCMPInterface = \{n : NAME \mid (cmpInterface\ n).intfState = enabled \bullet n\} \\
\forall c : (\mathrm{ran}\ cmpComputation) \bullet c.triggerInterface \subseteq (\mathrm{ran}\ cmpInterface)
\end{array}
$$

A component has initially got all of its interfaces disabled except *execInterface*.

InitComponent

$$
\begin{array}{l}
Component \\
\hline
\forall i : (\mathrm{ran}\ cmpInterface) \bullet \\
\quad i.intfState = disabled \\
opState = notloaded \\
cmpComputation = \emptyset
\end{array}
$$

PromoteChangeCMPInterfaceState

$$
\begin{array}{l}
\Delta Component \\
\Delta Interface \\
intfName? : NAME \\
\hline
intfName? \in (\mathrm{dom}\ cmpInterface) \\
cmpInterface' = cmpInterface \oplus \\
\quad \{intfName? \mapsto \theta Interface'\} \\
execInterface' = execInterface \\
opState' = opState \\
cmpComputation' = cmpComputation \\
measureInterface' = measureInterface
\end{array}
$$

ChangeComponentIntfState

$$
\begin{array}{l}
\widehat{=} \\
\quad \exists\, Interface \bullet \\
\quad ChangeInterfaceState \land \\
\quad PromoteChangeCMPInterfaceState
\end{array}
$$

ChangeComponentInternalState

$$
\begin{array}{l}
\Delta Component \\
newCMPState? : COMPONENTSTATE \\
\hline
opState \neq newCMPState? \\
opState' = newCMPState? \\
activeCMPInterface' = \\
\quad activeCMPInterface \\
execInterface' = execInterface \\
cmpInterface' = cmpInterface \\
cmpComputation' = \\
\quad cmpComputation \\
measureInterface' = \\
\quad measureInterface
\end{array}
$$

Changing a component's state is equivalent to changing its internal state or the state of one or more of its interfaces. *LoadComponent* and *ExecuteComponent* are

examples of operations that change the state of a component. If you only changing the interface state, then the values relating to ChangeInternal state are ignored.

EnableComponentExecution
ChangeComponentIntfState

$opState = loaded$

$execInterface \notin activeCMPInterface$

$newIntfState? = enabled$

$intfName? = execInterface$

DisableComponentExecution
ChangeComponentIntfState

$opState = loaded$

$execInterface \in activeCMPInterface$

$newIntfState? = disabled$

$intfName? = execInterface$

EnableComponentMeasurement
ChangeComponentIntfState

$opState = loaded$

$measureInterface \notin activeCMPInterface$

$newIntfState? = enabled$

$intfName? = measureInterface$

DisableComponentMeasurement
ChangeComponentIntfState

$opState = loaded$

$measureInterface \in activeCMPInterface$

$newIntfState? = disabled$

$intfName? = measureInterface$

LoadComponent
ChangeComponentInternalState

$opState = notloaded$

$newCMPState? = loaded$

UnloadComponent
ChangeComponentInternalState

$opState \in \{loaded, running\}$

$newCMPState? = notloaded$

ExecuteComponent
ChangeComponentInternalState

$opState = loaded$

$execInterface \in activeCMPInterface$

$newCMPState? = running$

StopComponentExecution
ChangeComponentInternalState

$opState = running$

$newCMPState? = loaded$

C.1.3 Connectors

A connector has multiple interfaces, some of which will be enabled at any given point. This state can be changed through the *ChangeConnectorState* operation.

Connector

$$cntProtocol : NAME \pfun PROTOCOL$$
$$cntInterface : NAME \pfun Interface$$
$$activeCNTInterface : \mathbb{P}\, NAME$$

$$activeCNTInterface = \{n : NAME;\ i : Interface \mid i = cntInterface\, n \land i.intfState = enabled \bullet n\}$$

InitConnector

$$Connector$$

$$\forall i : (\text{ran}\ cntInterface) \bullet i.intfState = disabled$$
$$cntProtocol = \emptyset$$

PromoteChangeCNTInterfaceState

$$\Delta Connector$$
$$\Delta Interface$$
$$intfName? : NAME$$

$$intfName? \in (\text{dom}\ cntInterface)$$
$$\theta Interface = cntInterface\, intfName?$$
$$cntInterface' = cntInterface \oplus \{intfName? \mapsto \theta Interface'\}$$
$$cntProtocol' = cntProtocol$$

$$ChangeConnectorIntfState \mathrel{\widehat=} \exists \Delta Interface \bullet ChangeInterfaceState \land PromoteChangeCNTInterfaceState$$

The set of active interfaces on a connector will automatically be updated based on the invariant on the *Connector* schema.

EnableConnectorInterface

$$ChangeConnectorIntfState$$

$$newIntfState? = enabled$$

DisableConnectorInterface

$$ChangeConnectorIntfState$$

$$newIntfState? = disabled$$

$$ChangeConnectorState \mathrel{\widehat=} EnableConnectorInterface \lor DisableConnectorInterface$$

C.1.4 Interactions

An interaction involves two or more components communicating over interfaces provided by a connector. It is created by connecting one end of a connector interface to an interface that exists on a component.

All the interfaces used in an interaction must be enabled. However, this requirement can only be enforced at the system level since an interaction has no direct access to the instances of connectors or components that participate in the specified interaction.

Interaction

$occursAt : Time$
$cntInstance : NAME$
$cmpIntfInstance : NAME \times Interface \rightarrowtail\!\!\!\!\!\!\rightarrow Interface$

$\#cmpIntfInstance \geq 2$

InitInteraction

$Interaction$

$occursAt = 0$

C.1.5 Basic System

A basic system comprises of components and connectors and has its state determined by the set of active components.

BasicSystem

$interaction : \mathbb{P}\, Interaction$
$sysconn : NAME \pfun Connector$
$syscomp : NAME \pfun Component$
$activeComponent : \mathbb{P}\, NAME$
$specialComponent : \mathbb{P}\, NAME$
$data : \mathbb{P}\, DATA;\ sysState : SYSTEMSTATE$

$activeComponent = \{n : NAME \mid (syscomp\ n).opState = running \bullet n\}$

$\forall x : interaction \bullet$
$x.cntInstance \in \mathrm{dom}\, sysconn \land$
$\mathrm{dom}(\mathrm{dom}\, x.cmpIntfInstance) \subseteq \mathrm{dom}\, syscomp$

InitBasicSystem

$BasicSystem$

$sysState = offline$

$interaction = \emptyset$

$\forall x : (\mathrm{dom}\, syscomp) \bullet$
$(syscomp\, x).opState = notloaded$

$data = \emptyset$

ChangeSystemOperationalState

$\Delta BasicSystem$
$newSystemState? : SYSTEMSTATE$

$sysState \neq newSystemState?$

$sysState' = newSystemState?$

$interaction' = interaction$

$specialComponent' = specialComponent$

$syscomp' = syscomp$

$sysconn' = sysconn$

$activeComponent' = activeComponent$

$data' = data$

PromoteComponentOperation

$\Delta BasicSystem$
$\Delta Component$
$cmpName? : NAME$

$sysState = live$

$cmpName? \in (\mathrm{dom}\, syscomp)$

$\theta Component = syscomp\ cmpName?$

$syscomp' = syscomp \oplus \{cmpName? \mapsto \theta Component'\}$

$specialComponent' = specialComponent$

$interaction' = interaction$

$sysconn' = sysconn$

$data' = data$

$sysState' = sysState$

Starting a system changes the system state to *live*. This will only occur if the system is not already *live*.

StartBasicSystem

$ChangeSystemOperationalState$

$newSystemState? = live$

StopBasicSystem

$ChangeSystemOperationalState$

$newSystemState? = offline$

$$ExecuteComponentBasicSystem \mathrel{\widehat=} \exists\, Component \bullet PromoteComponentOperation \land ExecuteComponent$$

$$LoadComponentBasicSystem \mathrel{\widehat=} \exists\, Component \bullet PromoteComponentOperation \land LoadComponent$$

$$EnableComponentExecutionBasicSystem \mathrel{\widehat=} \exists\, Component \bullet PromoteComponentOperation \land EnableComponentExecution$$

$$EnableComponentMeasurementBasicSystem \mathrel{\widehat=} \exists\, Component \bullet PromoteComponentOperation \land EnableComponentMeasurement$$

$$UnloadComponentBasicSystem \mathrel{\widehat=} \exists\, Component \bullet PromoteComponentOperation \land UnloadComponent$$

$$DisableComponentExecutionBasicSystem \mathrel{\widehat=} \exists\, Component \bullet PromoteComponentOperation \land DisableComponentExecution$$

$$DisableComponentMeasurementBasicSystem \mathrel{\widehat=} \exists\, Component \bullet PromoteComponentOperation \land DisableComponentMeasurement$$

$$StopComponentExecutionBasicSystem \mathrel{\widehat=} \exists\, Component \bullet PromoteComponentOperation \land StopComponentExecution$$

An interaction is created by passing the identities of the elements that participate in an interaction.

CreateInteractionBasicSystem

$\Delta BasicSystem$
$cmpName1?, cmpName2? : NAME$
$cmpIntf1?, cmpIntf2? : Interface$
$connName? : NAME$
$cntIntf? : Interface$
$occursAt? : Time$

$sysState = live$

$cmpName1? \in activeComponent$

$cmpName2? \in activeComponent$

$cmpIntf1? \in \mathrm{ran}((syscomp\ cmpName1?).cmpInterface)$

$cmpIntf2? \in \mathrm{ran}((syscomp\ cmpName2?).cmpInterface)$

$cmpIntf1?.intfState = enabled$

$cmpIntf1?.intfState = enabled$

$$interaction' = interaction \cup \{i : Interaction \mid i.occursAt = occursAt? \land i.cntInstance = connName? \land i.cmpIntfInstance = \{(cmpName1?, cmpIntf1?) \mapsto cntIntf?, (cmpName2?, cmpIntf2?) \mapsto cntIntf?\}\}$$

$syscomp' = syscomp$

$specialComponent' = specialComponent$

$sysconn' = sysconn$

$data' = data$

$sysState' = sysState$

TriggerComponentComputation

$\Delta Component$
$trigIntf? : NAME$
$compName? : NAME$

$opState = running$

$trigIntf? \in activeCMPInterface$

$execInterface' = execInterface$

$measureInterface' = measureInterface$

$cmpInterface' = cmpInterface$

$cmpComputation' = cmpComputation$

$$TriggerComponentComputationBasicSystem \mathrel{\widehat=} \exists\, Component \bullet PromoteComponentOperation \land TriggerComponentComputation$$

C.2 Protected Basic System

All protections are defined with a protection function, X, which specifies what components are protected from the given component, and a component, P, responsible for enforcing the protection.

ProtectionSystem

$BasicSystem$
$P : NAME$
$X : NAME \pfun \mathbb{P}\, NAME$

$P \in \mathrm{dom}\, syscomp$

$\forall A : NAME;\ B : \mathbb{P}\, NAME \bullet (A, B) \in X \Rightarrow$
$A \notin B \land A \in \mathrm{dom}\, syscomp \land B \subset \mathrm{dom}\, syscomp$

C.2.1 Protection through restricted interactions

The *RestrictedInteractionProtectionSystem* schema defines a protection function *restrictedInteraction* which contains all those components from the system (enforced by invariants on the generic function X defined in *ProtectionSystem*) that do not participate in the same interaction, with a given component.

RestrictedInteractionProtectionSystem

$$
\begin{array}{l}
BasicSystem \\
protector : NAME \\
restrictedInteraction : NAME \pfun \mathbb{P}\, NAME \\
\hline
ProtectionSystem[restrictedInteraction / X, protector / P] \\
\forall\, x, y : \mathrm{dom}\, syscomp;\ i : interaction;\ a, b : Interface \bullet \\
\quad y \in restrictedInteraction\, x \Leftrightarrow \\
\quad\quad (x, a) \in \mathrm{dom}\, i.cmpIntfInstance \land (y, b) \notin \mathrm{dom}\, i.cmpIntfInstance
\end{array}
$$

InitRIPS

$$
\begin{array}{l}
RestrictedInteractionProtectionSystem \\
\hline
InitBasicSystem
\end{array}
$$

An interaction can only be created if the two components are not restricted from interacting.

CreateInteractionRIPS

$$
\begin{array}{l}
\Delta RestrictedInteractionProtectionSystem \\
CreateInteractionBasicSystem \\
\hline
cmpName1? \notin restrictedInteraction\, cmpName2? \\
cmpName2? \notin restrictedInteraction\, cmpName1?
\end{array}
$$

C.2.2 Protection through memory isolation

A memory system is defined as consisting of a set of addresses and some content in each address.

MemorySystem

$$
\begin{array}{l}
address : \mathbb{P}\, ADDRESS \\
content : ADDRESS \to DATA \\
\hline
\mathrm{dom}\, content \subseteq address
\end{array}
$$

InitMemorySystem

$$
\begin{array}{l}
MemorySystem \\
\hline
content = \emptyset
\end{array}
$$

A generic operation on the memory system should not change the available set of addresses. All the other states do not change except that of the address specified.

Leaves the state of the memory system unchanged (i.e. both *content* and *address* does not change.

OperateOnMemorySystem

$\Delta MemorySystem$

$memAddress? : ADDRESS$

$memAddress? \in address$

$address' = address$

ReadMemoryState

OperateOnMemorySystem

$memContent! : DATA$

$memContent! = content\ memAddress?$

$content' = content$

ChangeMemoryState

OperateOnMemorySystem

$newcontent? : DATA$

$content' = content \oplus \{memAddress? \mapsto newcontent?\}$

The *MemoryProtectionSystem* states that the component referred to in the *MemorySystem* must be present in the *BasicSystem* and that any two components are not assigned the same addresses, but could share access to the same addresses.

MemoryProtectionSystem

BasicSystem

$memory : MemorySystem$

$memAccessCtr : NAME$

$access, assigned : NAME \pfun \mathbb{P}\, ADDRESS$

$memoryProtected : NAME \pfun \mathbb{P}\, NAME$

$ProtectionSystem[memoryProtected/X, memAccessCtr/P]$

$\mathrm{dom}\ access \subseteq \mathrm{dom}\ syscomp$

$\forall\, addressSet : \mathrm{ran}\ access \bullet addressSet \subseteq memory.address$

$\forall\, addressSet : \mathrm{ran}\ assigned \bullet addressSet \subseteq memory.address$

$\forall\, x, z : \mathrm{dom}\ syscomp;\ p : \mathrm{ran}\ memoryProtected \bullet$
$\quad assigned\ x \subseteq access\ x \land assigned\ z \cap assigned\ x = \emptyset$

A component requesting access to memory system will only be able to perform the operation if they have been given explicit access to the requested address.

InitMPS

MemoryProtectionSystem

InitBasicSystem

PromoteBSOperationMPS

$\Delta MemoryProtectionSystem$

$memAccessCtr' = memAccessCtr$

$access' = access$

$assigned' = assigned$

$memoryProtected' = memoryProtected$

$memory' = memory$

PromoteMemoryOperationToMPS

$\Delta MemoryProtectionSystem$
$\Delta MemorySystem$
$requestor? : NAME$
$memAddress? : ADDRESS$

$requestor? \in activeComponent$

$\theta MemorySystem = memory$

$memAddress? \in access\ requestor?$

$interaction' = interaction$

$data' = data$

$activeComponent' = activeComponent$

$specialComponent' = specialComponent$

$syscomp' = syscomp$

$sysconn' = sysconn$

$sysState' = sysState$

$memAccessCtr' = memAccessCtr$

$access' = access$

$assigned' = assigned$

$memoryProtected' = memoryProtected$

$$ChangeMemoryStateMPS \mathrel{\widehat=} \exists\, MemorySystem \bullet PromoteMemoryOperationToMPS \land ChangeMemoryState$$

$$ReadMemoryStateMPS \mathrel{\widehat=} \exists\, MemorySystem \bullet PromoteMemoryOperationToMPS \land ReadMemoryState$$

StartSystemMPS

$StartBasicSystem$
$PromoteBSOperationMPS$

EnableComponentExecutionMPS

$EnableComponentExecutionBasicSystem$
$PromoteBSOperationMPS$

StopSystemMPS

$StopBasicSystem$
$PromoteBSOperationMPS$

EnableComponentMeasurementMPS

$EnableComponentMeasurementBasicSystem$
$PromoteBSOperationMPS$

LoadComponentMPS

$$\begin{array}{l} \textit{LoadComponentBasicSystem} \\ \textit{PromoteBSOperationMPS} \\ \hline memory' = memory \end{array}$$

UnloadComponentMPS

$$\begin{array}{l} \textit{UnloadComponentBasicSystem} \\ \textit{PromoteBSOperationMPS} \end{array}$$

ExecuteComponentMPS

$$\begin{array}{l} \textit{ExecuteComponentBasicSystem} \\ \textit{PromoteBSOperationMPS} \end{array}$$

Access to a given address can be granted or revoked by changing the *access* relation.

MemoryAccessOperation

$$\begin{array}{l} \Delta \textit{MemoryProtectionSystem} \\ \hline sysState = live \\ interaction' = interaction \\ syscomp' = syscomp \\ sysState' = sysState \\ sysconn' = sysconn \\ activeComponent' = activeComponent \\ specialComponent' = specialComponent \\ data' = data \\ sysState' = sysState \\ memory' = memory \\ memAccessCtr' = memAccessCtr \\ memoryProtected' = memoryProtected \end{array}$$

GrantAccessMPS

$$\begin{array}{l} \textit{MemoryAccessOperation} \\ grantAdd? : ADDRESS;\ grantCMP? : NAME \\ inmem? : NAME \\ \hline grantCMP? \in \mathrm{dom}\ syscomp \\ grantAdd? \in memory.address \\ access' = access \oplus \{grantCMP? \mapsto access\ grantCMP? \cup \{grantAdd?\}\} \\ assigned' = assigned \end{array}$$

RevokeAccess

$$
\begin{array}{l}
MemoryAccessOperation \\
revokeAdd? : ADDRESS;\ revokeCMP? : NAME \\
\hline
revokeCMP? \in \mathrm{dom}\ syscomp \\
revokeAdd? \in memory.address \\
access' = access \oplus \{revokeCMP? \mapsto access\ revokeCMP? \setminus \{revokeAdd?\}\} \\
assigned' = assigned
\end{array}
$$

Assigning an address to a component also implies granting that component access to that address.

AssignAddress

$$
\begin{array}{l}
MemoryAccessOperation \\
grantAdd? : ADDRESS;\ grantCMP? : NAME \\
\hline
grantCMP? \in \mathrm{dom}\ syscomp \\
grantAdd? \in memory.address \\
access' = access \oplus \{grantCMP? \mapsto access\ grantCMP? \cup \{grantAdd?\}\} \\
\forall\, s : \mathrm{ran}\ assigned \bullet grantAdd? \notin s \\
assigned' = assigned \oplus \{grantCMP? \mapsto (assigned\ grantCMP? \cup \{grantAdd?\})\}
\end{array}
$$

UnassignAddress

$$
\begin{array}{l}
MemoryAccessOperation \\
revokeAdd? : ADDRESS;\ revokeCMP? : NAME \\
\hline
revokeCMP? \in \mathrm{dom}\ syscomp \\
revokeAdd? \in memory.address \\
access' = access \oplus \{revokeCMP? \mapsto access\ revokeCMP? \setminus \{revokeAdd?\}\} \\
revokeAdd? \in assigned\ revokeCMP? \\
assigned' = assigned \oplus \{revokeCMP? \mapsto assigned\ revokeCMP? \setminus \{revokeAdd?\}\}
\end{array}
$$

C.2.3 Protection through ring architecture

Protection based on ring architecture is defined in terms of the ring level to which each component is assigned.

RingArchitectureProtectionSystem

$$
\begin{array}{l}
BasicSystem \\
cpuSubsystem : NAME \\
privilege : NAME \pfun Ring \\
ringProtected : NAME \pfun \mathbb{P}\, NAME \\
\hline
ProtectionSystem[ringProtected/X, cpuSubsystem/P] \\
\dom privilege \subseteq \dom syscomp \\
\forall\, x : \dom syscomp \bullet \\
\quad ringProtected\ x = \{y : \dom syscomp \mid privilege\ x < privilege\ y\}
\end{array}
$$

InitRAPS

$$
\begin{array}{l}
RingArchitectureProtectionSystem \\
\hline
InitBasicSystem
\end{array}
$$

PromoteBasicSystemOpToRAPS

$$
\begin{array}{l}
\Delta RingArchitectureProtectionSystem \\
\hline
cpuSubsystem' = cpuSubsystem \\
privilege' = privilege \\
ringProtected' = ringProtected
\end{array}
$$

StartSystemRAPS

$$
\begin{array}{l}
StartBasicSystem \\
PromoteBasicSystemOpToRAPS
\end{array}
$$

StopSystemRAPS

$$
\begin{array}{l}
StopBasicSystem \\
PromoteBasicSystemOpToRAPS
\end{array}
$$

LoadComponentRAPS

$$
\begin{array}{l}
LoadComponentBasicSystem \\
PromoteBasicSystemOpToRAPS
\end{array}
$$

UnloadComponentRAPS

$$
\begin{array}{l}
UnloadComponentBasicSystem \\
PromoteBasicSystemOpToRAPS
\end{array}
$$

ExecuteComponentRAPS

$$
\begin{array}{l}
ExecuteComponentBasicSystem \\
PromoteBasicSystemOpToRAPS
\end{array}
$$

EnableComponentExecutionRAPS

$$
\begin{array}{l}
EnableComponentExecutionBasicSystem \\
PromoteBasicSystemOpToRAPS
\end{array}
$$

EnableComponentMeasurementRAPS

$$
\begin{array}{l}
EnableComponentMeasurementBasicSystem \\
PromoteBasicSystemOpToRAPS
\end{array}
$$

AssignPrivilege

$\Delta RingArchitectureProtectionSystem$
$assignToCMP? : NAME;\ newRingLevel? : Ring$

$sysState = live$

$privilege' = privilege \oplus \{assignToCMP? \mapsto newRingLevel?\}$

$interaction' = interaction$

$syscomp' = syscomp$

$sysconn' = sysconn$

$activeComponent' = activeComponent$

$data' = data$

$sysState' = sysState$

$cpuSubsystem' = cpuSubsystem$

$ringProtected' = ringProtected$

C.2.4 Combining the Protection Mechanisms

A protected basic system combines *RingArchitectureProtectionSystem*, *MemoryProtectionSystem* and *RestrictedInteractionProtectionSystem*.

$$ProtectedBasicSystem \mathrel{\widehat{=}} RingArchitectureProtectionSystem \land MemoryProtectionSystem \land RestrictedInteractionProtectionSystem$$

InitPBS

$ProtectedBasicSystem$

$InitMPS \land InitRAPS \land InitRIPS$

PromoteBSOperationPBS

$\Delta ProtectedBasicSystem$

$sysconn' = sysconn$

$syscomp' = syscomp$

$data' = data$

$sysState' = sysState$

$memory' = memory$

$memAccessCtr' = memAccessCtr$

$access' = access$

$assigned' = assigned$

$memoryProtected' = memoryProtected$

$cpuSubsystem' = cpuSubsystem$

$privilege' = privilege$

$ringProtected' = ringProtected$

PromoteMPSOperationPBS
$\Delta ProtectedBasicSystem$

$interaction' = interaction$
$sysconn' = sysconn$
$syscomp' = syscomp$
$activeComponent' = activeComponent$
$data' = data$
$sysState' = sysState$
$memory' = memory$
$memAccessCtr' = memAccessCtr$
$memoryProtected' = memoryProtected$
$cpuSubsystem' = cpuSubsystem$
$privilege' = privilege$
$ringProtected' = ringProtected$

PromoteRPSOperationPBS
$\Delta ProtectedBasicSystem$

$interaction' = interaction$
$sysconn' = sysconn$
$syscomp' = syscomp$
$activeComponent' = activeComponent$
$data' = data$
$sysState' = sysState$
$memory' = memory$
$memAccessCtr' = memAccessCtr$
$access' = access$
$assigned' = assigned$
$memoryProtected' = memoryProtected$
$cpuSubsystem' = cpuSubsystem$

StartSystemPBS
$\Delta ProtectedBasicSystem$
$StartSystemRAPS$
$StartSystemMPS$

ExecuteComponentPBS
$\Delta ProtectedBasicSystem$
$ExecuteComponentRAPS$
$ExecuteComponentMPS$

StopSystemPBS
$\Delta ProtectedBasicSystem$
$StopSystemRAPS$
$StopSystemMPS$

EnableComponentExecutionPBS
$\Delta ProtectedBasicSystem$
$EnableComponentExecutionRAPS$
$EnableComponentExecutionMPS$

LoadComponentPBS
$\Delta ProtectedBasicSystem$
$LoadComponentRAPS$
$LoadComponentMPS$

EnableComponentMeasurementPBS
$\Delta ProtectedBasicSystem$
$EnableComponentMeasurementRAPS$
$EnableComponentMeasurementMPS$

UnloadComponentPBS
$\Delta ProtectedBasicSystem$
$UnloadComponentMPS$
$UnloadComponentRAPS$

AssignPrivilegePBS
$AssignPrivilege$
$PromoteRPSOperationPBS$

GrantAccessPBS

GrantAccessMPS
PromoteMPSOperationPBS

CreateInteractionPBS

PromoteBSOperationPBS
CreateInteractionRIPS

RevokeAccessPBS

RevokeAccess
PromoteMPSOperationPBS

C.2.5 Shared Resources

$ASMP_1$: `Shared resources` — any resources that may be shared by components on the system is assumed to be free from vulnerabilities.

SharedResourceSystem

$$\begin{array}{l} ProtectedBasicSystem \\ VulnerabilityFree, SharedResource : \mathbb{P}\, NAME \\ \hline VulnerabilityFree \subseteq \mathrm{dom}\, syscomp \\ SharedResource \subseteq VulnerabilityFree \end{array}$$

C.2.6 Direct Memory Access (DMA)

A DMA enabled system is defined as a protected basic system which has a set of components that are capable of accessing memory directly.

DMAEnabledProtectedBasicSystem

$$\begin{array}{l} SharedResourceSystem \\ dmaCapableDevices : \mathbb{P}\, NAME \\ \hline dmaCapableDevices \subseteq \mathrm{dom}\, syscomp \end{array}$$

$PROP_2$: `DMA` — any device included in the set of DMA capable devices has *implicit* access to all memory addresses on the system

DMAAccessibleMemory

$$\begin{array}{l} DMAEnabledProtectedBasicSystem \\ \hline \forall\, dev : dmaCapableDevices \bullet memory.address \subset (access\ dev) \end{array}$$

StartSystemDMAPBS

$$\begin{array}{l} \Delta DMAEnabledProtectedBasicSystem \\ StartSystemPBS \\ \hline access' = access \cup \{ dev : \mathrm{dom}\, syscomp;\ a : \mathbb{P}\, ADDRESS \mid \\ \qquad dev \in dmaCapableDevices \land a \subseteq memory.address \} \end{array}$$

InitDMAPBS

DMAEnabledProtectedBasicSystem

InitPBS

C.2.7 Ring 0 Memory Access

$PROP_3$: *Ring 0 memory address* — any component that is running in a higher privilege has access to all the memory used by a component running in a lower privilege. Consequently, any component running in *Ring 0* has access to all of memory.

*Ring*0*MemoryAccess*

ProtectedBasicSystem

$$\forall x, y : \mathrm{dom}\ syscomp \bullet (privilege\ x < privilege\ y \lor privilege\ x = 0) \Rightarrow (assigned\ y \subseteq access\ x)$$

C.3 Trust Enablers

Components that enable system trustworthiness are formalised in this section.

C.3.1 Integrity Measurement

An integrity value comprises of the identity, a set of interfaces and the computations that are available for a given component. This could be enhanced with a set of active interfaces as well as the state of the component at the time of recording this value.

IntegrityValue

$identity : NAME$
$interface : \mathbb{P}\ Interface$
$computation : \mathbb{P}\ Computation$

EnhancedIntegrityValue

$IntegrityValue$
$activeInterfaces : \mathbb{P}\ Interface$
$state : COMPONENTSTATE$

IntegrityMeasurementService

$service : Component$
$recordInterface : NAME$

$$recordInterface \in \mathrm{dom}\ service.cmpInterface$$

IntegrityMeasurableSystem

$DMAEnabledProtectedBasicSystem$

$TCB : \mathbb{P}\, NAME$

$ims : NAME \pfun IntegrityMeasurementService$

$log : NAME \pfun \mathbb{P}\, IntegrityValue$

$TCB \subseteq \mathrm{dom}\, syscomp$

$\forall\, c : \mathrm{dom}\, ims \bullet c \in TCB \land (ims\ c).service \in \mathrm{ran}\, syscomp$

$\mathrm{dom}\, log \subseteq \mathrm{dom}\, ims$

InitIMS

$IntegrityMeasurableSystem$

$InitDMAPBS$

$log = \emptyset$

StartSystemIMS

$\Delta IntegrityMeasurableSystem$

$StartSystemDMAPBS$

StopSystemIMS

$\Delta IntegrityMeasurableSystem$

$StopSystemPBS$

LoadComponentIMS

$\Delta IntegrityMeasurableSystem$

$LoadComponentPBS$

UnloadComponentIMS

$\Delta IntegrityMeasurableSystem$

$UnloadComponentPBS$

ExecuteComponentIMS

$\Delta IntegrityMeasurableSystem$

$ExecuteComponentPBS$

EnableComponentExecutionIMS

$\Delta IntegrityMeasurableSystem$

$EnableComponentExecutionPBS$

EnableComponentMeasurementIMS

$\Delta IntegrityMeasurableSystem$

$EnableComponentMeasurementPBS$

AssignPrivilegeIMS

$\Delta IntegrityMeasurableSystem$

$AssignPrivilegePBS$

GrantAccessIMS

$\Delta IntegrityMeasurableSystem$

$GrantAccessPBS$

RevokeAccessIMS

$\Delta IntegrityMeasurableSystem$

$RevokeAccessPBS$

RecordComponentIntegrityToIMS

$$
\begin{array}{l}
cmpToMeasure? : NAME \\
imsComponent? : NAME \\
\Xi IntegrityMeasurementService \\
\Delta IntegrityMeasurableSystem \\
\hline
sysState = live \\
cmpToMeasure? \in \mathrm{dom}\, syscomp \\
recordInterface \in service.activeCMPInterface \\
imsComponent? \in \mathrm{dom}\, ims \\
\theta IntegrityMeasurementService = ims\ imsComponent? \\
(syscomp\ cmpToMeasure?).measureInterface \in \\
\quad (syscomp\ cmpToMeasure?).activeCMPInterface \\
log' = log \oplus \{imsComponent? \mapsto (log\ imsComponent? \cup \{i : IntegrityValue \mid \\
\quad i.identity = cmpToMeasure? \land \\
\quad i.interface = \mathrm{ran}((syscomp\ cmpToMeasure?).cmpInterface) \land \\
\quad i.computation = \mathrm{ran}((syscomp\ cmpToMeasure?).cmpComputation)\})\}
\end{array}
$$

MeasurementComplete

$$
\begin{array}{l}
IntegrityMeasurableSystem \\
\hline
\forall c : TCB \bullet (\exists\, RecordComponentIntegrityToIMS \bullet cmpToMeasure? = c)
\end{array}
$$

MeasuredBeforeExecute

$$
\begin{array}{l}
IntegrityMeasurableSystem \\
\hline
\forall c : \mathrm{dom}\, syscomp \bullet \\
\quad (\exists e, m : Interaction;\ RecordComponentIntegrityToIMS \bullet \\
\qquad e \in interaction \land m \in interaction \land \\
\qquad m.occursAt < e.occursAt \land cmpToMeasure? = c \land \\
\qquad (\exists p : \mathrm{dom}(m.cmpIntfInstance) \bullet \\
\qquad\quad (c, ((syscomp\ c).cmpInterface)\ (syscomp\ c).measureInterface) \\
\qquad\qquad = p) \land \\
\qquad (\exists p : \mathrm{dom}(e.cmpIntfInstance) \bullet \\
\qquad\quad (c, ((syscomp\ c).cmpInterface)\ (syscomp\ c).execInterface) = p))
\end{array}
$$

C.3.2 Attestation

We define attestation as a combination of some secret data, such as private key and integrity values recorded by one or more integrity measurement services.

Attestation

$secret : DATA$

$log : NAME \pfun \mathbb{P}\, IntegrityValue$

AttestationService

$service : Component$

$attestInterface : Interface$

$secret : DATA$

$attestInterface \in (\mathrm{ran}(service.cmpInterface))$

AttestableSystem

IntegrityMeasurableSystem

$ats : NAME \pfun AttestationService$

$\forall x : \mathrm{ran}\ ats \bullet x.service \in \mathrm{ran}\ syscomp$

InitATS

AttestableSystem

InitIMS

StartSystemATS

$\Delta AttestableSystem$

StartSystemIMS

StopSystemATS

$\Delta AttestableSystem$

StopSystemIMS

LoadComponentATS

$\Delta AttestableSystem$

LoadComponentIMS

UnloadComponentATS

$\Delta AttestableSystem$

UnloadComponentIMS

ExecuteComponentATS

$\Delta AttestableSystem$

ExecuteComponentIMS

EnableComponentExecutionATS

$\Delta AttestableSystem$

EnableComponentExecutionIMS

EnableComponentMeasurementATS

$\Delta AttestableSystem$

EnableComponentMeasurementIMS

AssignPrivilegeATS

$\Delta AttestableSystem$

AssignPrivilegeIMS

GrantAccessATS

$\Delta AttestableSystem$

GrantAccessIMS

RevokeAccessATS

$\Delta AttestableSystem$

RevokeAccessIMS

AttestSystem

$\Delta AttestableSystem$
$atsName? : NAME$
$attestations! : Attestation$

$sysState = live$

$(ats\ atsName?).service = syscomp\ atsName?$

$atsName? \in activeComponent$

$attestations!.secret = (ats\ atsName?).secret$

$attestations!.log = log$

$\theta AttestableSystem' = \theta AttestableSystem$

IsolatedAttestationService

AttestableSystem

$$\forall\, a : \mathrm{dom}\ ats;\ c : \mathrm{dom}\ syscomp \bullet (c \notin TCB) \Leftrightarrow ((c \in restrictedInteraction\ a) \land (c \in ringProtected\ a) \land (c \in memoryProtected\ a))$$

$PROP_5$: `Report completeness` — attestations will be meaningful if the attestation reports are complete. We define *ReportCompleteness* to capture this property, which specifies that every program whose integrity was measured was also reported.

ReportCompleteness

AttestableSystem

$$\forall\, RecordComponentIntegrityToIMS \mid imsComponent? \in \mathrm{dom}\ ims \bullet (\exists\, AttestSystem \bullet atsName? \in \mathrm{dom}\ ats \land imsComponent? \in \mathrm{dom}\ attestations!.log)$$

ReportConsistence

AttestableSystem

$$\forall\, AttestSystem \mid atsName? \in \mathrm{dom}\ ats \bullet attestations!.log \subseteq log \land (\exists\, RecordComponentIntegrityToIMS \bullet imsComponent? \in \mathrm{dom}\ attestations!.log)$$

StorageService

$service : Component$

$keyaccess : KEY \leftrightarrow NAME$

SecureStorageSystem

$ProtectedBasicSystem$

$storage : NAME \pfun StorageService$

$\forall\, c : \operatorname{ran} storage \bullet$
$\quad c.service \in \operatorname{ran} syscomp$

InitSSS

$SecureStorageSystem$

$InitPBS$

EnableComponentExecutionSSS

$\Delta SecureStorageSystem$

$EnableComponentExecutionPBS$

StartSystemSSS

$\Delta SecureStorageSystem$

$StartSystemPBS$

EnableComponentMeasurementSSS

$\Delta SecureStorageSystem$

$EnableComponentMeasurementPBS$

StopSystemSSS

$\Delta SecureStorageSystem$

$StopSystemPBS$

AssignPrivilegeSSS

$\Delta SecureStorageSystem$

$AssignPrivilegePBS$

LoadComponentSSS

$\Delta SecureStorageSystem$

$LoadComponentPBS$

GrantAccessSSS

$\Delta SecureStorageSystem$

$GrantAccessPBS$

UnloadComponentSSS

$\Delta SecureStorageSystem$

$UnloadComponentPBS$

ExecuteComponentSSS

$\Delta SecureStorageSystem$

$ExecuteComponentPBS$

RevokeAccessSSS

$\Delta SecureStorageSystem$

$RevokeAccessPBS$

StoreSecurely

$\Delta StorageService$

$dataIn?, dataOut! : DATA$

$key? : KEY;\ storingCMP? : NAME$

$service.opState = running$

$keyaccess' = keyaccess \oplus$
$\quad \{key? \mapsto storingCMP?\}$

$service' = service$

$StoreSecurelySSS \mathrel{\widehat=} \exists\, StorageService \bullet$
$\quad PromoteStorageOp \land StoreSecurely$

PromoteStorageOp

$\Delta StorageService$
$\Delta SecureStorageSystem$
$strName? : NAME$

$sysState = live$

$strName? \in \text{dom}\, syscomp$

$service \in \text{ran}\, syscomp$

$activeComponent' = activeComponent$

$syscomp' = syscomp$

$sysconn' = sysconn$

$interaction' = interaction$

$PROP_7$: `Secure secrets` — sensitive information is protected such that it can only be reviewed when the policy specified at the point of storing it is satisfied.

SecuredSecret

$SecureStorageSystem$

$$\forall x, y : (\text{dom}\, syscomp) \bullet \\ y \in (restrictedInteraction\, x) \land y \in (ringProtected\, x) \land \\ y \in (memoryProtected\, x) \Rightarrow \\ (\forall StoreSecurely \bullet (storingCMP? = x) \Rightarrow y = (keyaccess\, key?))$$

C.3.3 Trusted Isolation

A *TrustedIsolationSystem* is defined as an *AttestableSystem* that has all protectors defined for each protection mechanism are in the TCB and with their integrity recorded.

TrustedIsolationSystem

$AttestableSystem$

$\{protector, memAccessCtr, cpuSubsystem\} \subseteq TCB$

$(\exists RecordComponentIntegrityToIMS \bullet cmpToMeasure? = protector)$

$(\exists RecordComponentIntegrityToIMS \bullet cmpToMeasure? = memAccessCtr)$

$(\exists RecordComponentIntegrityToIMS \bullet cmpToMeasure? = cpuSubsystem)$

C.3.4 Secure I/O

A SecureIOSystem is a type of system that provides both trusted paths and channels by ensuring that all entities involved in every interaction can be surely identified.

SecureIOSystem

$$
\begin{array}{l}
\textit{IntegrityMeasurableSystem} \\
\hline
\forall\, i : \textit{interaction} \bullet \\
\quad (\forall\, p : \operatorname{dom} i.\textit{cmpIntfInstance} \bullet \\
\quad\quad (\exists\, \textit{RecordComponentIntegrityToIMS};\ \textit{Interface} \bullet \\
\quad\quad\quad (\textit{cmpToMeasure}?, \theta \textit{Interface}) = p))
\end{array}
$$

C.3.5 Secure Roots of Trust

The *TPM* has several keys and keeps a composite value of integrity measurements recorded on a system. The PCR structure is simplified. Ideally, this would be a set of sets.

TPM

$$
\begin{array}{l}
\textit{SRK}, \textit{AIK} : \textit{KEY} \\
\textit{PCR} : \textit{IntegrityValue} \\
\textit{extendIntf}, \textit{sealIntf}, \textit{unsealIntf}, \textit{quoteIntf} : \textit{Interface}
\end{array}
$$

The hash function is defined as a mapping of two integrity values to one

$$\textit{hash} : \textit{IntegrityValue} \times \textit{IntegrityValue} \nrightarrow \textit{IntegrityValue}$$

Extend

$$
\begin{array}{l}
\Delta \textit{TPM} \\
\textit{measurement}? : \textit{IntegrityValue} \\
\hline
\textit{extendIntf}.\textit{intfState} = \textit{enabled} \\
\textit{PCR}' = \textit{hash}\,(\textit{PCR}, \textit{measurement}?) \\
\textit{SRK}' = \textit{SRK} \\
\textit{AIK}' = \textit{AIK} \\
\textit{sealIntf}' = \textit{sealIntf} \\
\textit{unsealIntf}' = \textit{unsealIntf} \\
\textit{quoteIntf}' = \textit{quoteIntf} \\
\textit{extendIntf}' = \textit{extendIntf}
\end{array}
$$

Seal

$$
\begin{array}{l}
\Xi \textit{TPM} \\
d? : \textit{DATA}; \\
\textit{sealed_d}! : \textit{DATA} \times \textit{IntegrityValue} \\
\hline
\textit{sealIntf}.\textit{intfState} = \textit{enabled} \\
\textit{sealed_d}! = (d?, \textit{PCR}) \\
\theta \textit{TPM}' = \theta \textit{TPM}
\end{array}
$$

Unseal

ΞTPM
$d! : DATA;$
$sealed_d? : DATA \rightarrow IntegrityValue$

$unsealIntf.intfState = enabled$

$PCR \in \operatorname{ran} sealed_d?$

$d! = sealed_d?^{\sim}\ PCR$

$\theta TPM' = \theta TPM$

Quote

ΞTPM
$nonce? : NONCE$
$quote! : IntegrityValue \times$
$\quad NONCE \times KEY$

$quoteIntf.intfState = enabled$

$quote! = (PCR, nonce?, AIK)$

$\theta TPM' = \theta TPM$

The core root of trust for measurement (CRTM) has capabilities to perform self measurements (i.e. record its own integrity).

CRTM

$selfTest : Computation$
$measureIntf : Interface$

A *SecureRootofTrustSystem* has its roots of trust embedded in the TPM and CRTM.

SecureRootofTrustSystem

$ProtectedBasicSystem$
$tpm : NAME \rightarrowtail TPM$
$crtm : NAME \rightarrowtail CRTM$
$rootOfTrust : \mathbb{P}\ NAME$
$rootOfTrustFor : NAME \leftrightarrow Root$

$rootOfTrustFor = \{x : \operatorname{dom} tpm;\ r : \{Reporting, Storage\} \bullet x \mapsto r\} \cup$
$\quad \{y : \operatorname{dom} crtm;\ r : \{Measurement\} \bullet y \mapsto r\}$

$rootOfTrust = \operatorname{dom} rootOfTrustFor$

PromoteTPMOperationToSRS

ΔTPM
$tpmName? : NAME$
$\Delta SecureRootofTrustSystem$

$tpm\ tpmName? = \theta TPM$

$tpm' = tpm \oplus \{tpmName? \mapsto \theta TPM'\}$

$$QuoteSRS \mathrel{\widehat=} \exists\, TPM \bullet PromoteTPMOperationToSRS \land Quote$$

$$UnsealSRS \mathrel{\widehat=} \exists\, TPM \bullet PromoteTPMOperationToSRS \land Unseal$$

$$ExtendSRS \mathrel{\widehat=} \exists\, TPM \bullet PromoteTPMOperationToSRS \land Extend$$

$$RecordComponentIntegritySRS \mathrel{\widehat=} RecordComponentIntegrityToIMS \land ExtendSRS$$

$$SealSRS \mathrel{\widehat=} \exists\, TPM \bullet PromoteTPMOperationToSRS \land Seal$$

$$AttestSystemSRS \mathrel{\widehat=} AttestSystem \land QuoteSRS$$

C.4 Trustworthy System Model

A *TrustworthySystem* is a composition of various subsystems defined above, all of which build on the *ProtectedBasicSystem* model. It inherits properties from its subsystems and provides a number of operations which are fulfiled by one or a combination of its subsystems. A *TrustworthySystem* requires that all components that provide attestation, integrity measurement and secure storage are protected by one of the roots of trust.

TrustworthySystem

$$SecureIOSystem \land AttestableSystem \land SecureRootofTrustSystem \land SecureStorageSystem \land IntegrityMeasurableSystem$$

$$DMAAccessibleMemory$$

$$Ring0MemoryAccess$$

$$SecuredSecret$$

$$IsolatedAttestationService$$

$$MeasuredBeforeExecute$$

$$MeasurementComplete$$

InitTrustworthySystem

$$TrustworthySystem$$

$$InitIMS \land InitATS \land InitSSS$$

$$interaction = \emptyset$$

$$data = \emptyset$$

EnableCMPExecutionTrustworthySystem
$\Delta TrustworthySystem$
EnableComponentExecutionSSS
EnableComponentExecutionIMS
EnableComponentExecutionATS

LoadCMPTrustworthySystem
$\Delta TrustworthySystem$
LoadComponentSSS
LoadComponentIMS
LoadComponentATS

EnableCMPMeasurementTrustworthySystem
$\Delta TrustworthySystem$
EnableComponentMeasurementSSS
EnableComponentMeasurementIMS
EnableComponentMeasurementATS

UnloadCMPTrustworthySystem
$\Delta TrustworthySystem$
UnloadComponentSSS
UnloadComponentIMS
UnloadComponentATS

StartTrustworthySystem
$\Delta TrustworthySystem$
StartSystemSSS
StartSystemIMS
StartSystemATS

ExecuteCMPTrustworthySystem
$\Delta TrustworthySystem$
ExecuteComponentSSS
ExecuteComponentIMS
ExecuteComponentATS

StopTrustworthySystem
$\Delta TrustworthySystem$
StopSystemSSS
StopSystemIMS
StopSystemATS

RecordCMPIntegrityTrustworthySystem
$\Delta TrustworthySystem$
RecordComponentIntegritySRS

RevokeAccessTrustworthySystem
$\Delta TrustworthySystem$
RevokeAccessPBS
RevokeAccessPBS
RevokeAccessPBS

AttestTrustworthySystem
$\Delta TrustworthySystem$
AttestSystemSRS

StoreSecurelyTrustworthySystem
$\Delta TrustworthySystem$
StoreSecurelySSS

AssignPrivilegeTrustworthySystem
$\Delta TrustworthySystem$
AssignPrivilegeSSS

CreateInteractionTrustworthySystem
$\Delta TrustworthySystem$
CreateInteractionRIPS

GrantAccessTrustworthySystem
$\Delta TrustworthySystem$
GrantAccessPBS
GrantAccessPBS
GrantAccessPBS

Init
InitTrustworthySystem

Appendix D

Complete Abstract Model Using CSP

This chapter presents the full CSP abstract model discussed in Chapter 5.

```
{-
The following are the data types used in the rest of the model.
-}
datatype Data = --Roles
                btl| bios| os| pep | app | mdw | app_man | s3 | ima | rs | mal |
                -- Actual Code
                grub_4_2 | linux_2_6 | ima1_2 | ats_5 | rserv_1_8 | appman1_2 |
                gt_5 | jsm_1_9 | snstv_job | eveware_5x | amibios |
                --ExternalAgents
                V |
                --Identities
                M |
                --Address
                l0 | l1 |  l2 | l3 | l4 | l5 | l6 | l7 |l8 | l9 | l10 |
                --data that can be stored
                secret |
                --Nonces
                n |
                --Quotes
                Good | Bad |
                --blobs
                blob |
                --Results, possibly use good bad or garbage
                R1 | R2 |
                --Garbage data
                Garbage |
                --Encryption data
                Encrypt.(ALL_KEYS,Seq(Data)) |
                --Keys
                K |
                -- Cooper's data types
                j1 |
                --Some groupings
                srvc | apps

Role            = {btl,bios,os,pep,app,mdw,app_man,s3,ima,rs}
Code            = {amibios,grub_4_2,linux_2_6,ima1_2,ats_5,rserv_1_8,appman1_2,
                       gt_5,jsm_1_9,snstv_job,eveware_5x}
Address         = {l0,l1,l2,l3,l4,l5,l6,l7,l8,l9,l10}
Identity        = {M}
ExternalAgent   = {V}
Secret          = {secret}
```

```
Storage          = {s3}
Nonce            = {n}
Quote            = {Good,Bad}
Blob             = {blob,Garbage}
Result           = {R1,Garbage}
Unsealable       = {Good, Garbage}
{-
Channels are defined to use one or more of the data types defined above.
-}
channel  config_bios,config_btl,config:Code.Role.Address
channel  config_os:Code.Address.Address.Address.Address

channel config_apps:Code.Address.Address.Address
channel init
channel selftest:Role.Code
channel read,write: Identity.Address.Code
channel load:Code.Role.Code
channel stop,extend:Code
channel run:Role.Code
channel store:Storage.Code.Secret
channel seal:Secret.Code.Blob
channel stored:Storage.Code.Blob
channel retrieve:Storage.Code.Blob
channel unseal:Blob.Secret.Unsealable
channel retrieved:Storage.Code.Secret
channel challenge:Nonce
channel quote:Nonce.Quote
channel response:Quote

--ABSTRACT MODEL
{-
A platform is combination of components and resources with each component's
behaviour constrained by the configuration. This means that platform will behave
according to what components exists and the configuration of those components.

-}
Platform(PlatformCfg,ResourceCfg,App_Runs,App_Interfaces)=
(
        (
        Runtime_System [|resource_interface|]
        Resources(ResourceCfg)
        )
        [|service_interface|]
        TestApps
        --Applications(App_Runs,App_Interfaces)
)
[|config_interface|]
PlatformCfg

resource_interface = Union({{|read,seal,unseal,quote,write|},{extend.x | x ← Code}})
config_interface = {|config|}
Run_Interface = {|run.rol.x | x← ProtocolElements,
        rol ← diff(PotentialCodeRole,{s3,rs,btl.os,app_man,ima})|}
App_Interface_Resource = {|challenge, response|}
service_interface =Union({App_Interface_Resource,Run_Interface})

--Runtime System
PotentialBtlLoc = {l1,l10}
PotentialBtlCode= {grub_4_2,eveware_5x}
PotentialOSLoc  = {l2,l10}
PotentialOSCode = {linux_2_6,eveware_5x}

Runtime_System  =
        init → selftest.bios.amibios →
        (□ loc : PotentialBtlLoc • config.amibios.btl.loc →
        (□ code: PotentialBtlCode • read.M.loc.code →
        load.amibios.btl.code → extend.code → BTL(code)))
```

```
BTL(id) =
        run.btl.id → (
        □ osloc : PotentialOSLoc •
        config.id.os.osloc → (□ code: PotentialOSCode • read.M.osloc.code →
        load.id.os.code → extend.code → Operating_System(code)))

Operating_System(id) =
        run.os.id → (□ apmloc: PotentialAPMLoc •
        config.id.app_man.apmloc → (□ code: PotentialAPManCode •
        read.M.apmloc.code → load.id.app_man.code → extend.code →
        run.app_man.code → (
        □ loc : PotentialTSSrvcLoc •
                config.code.ima.loc → (□ acode: PotentialTrustService • read.M.loc.acode →
                load.code.ima.acode → extend.acode → (
                run.ima.acode → AppManager'(code,id,{acode})
                [|ts_load_interface|]
                Trust_Services))
        )))

{-Interaces-}
app_load_interface = {||}
ts_load_interface = {|run,load|}
                        {-

ts_load_interface = {|run.x.ts,load.appman1_2.app_man.y
                        | x ← diff(PotentialCodeRole,{ima}),
                        y ← PotentialCode2Run,
                        ts ← PotentialTrustService|}-}
PotentialAPMLoc = {13,110}
PotentialImaLoc = {14,110}
PotentialRsLoc = {15,110}
PotentialS3Loc = {16,110}
PotentialJob1Loc = {18,110}
PotentialJob2Loc = {19,110}
PotentialMdwLoc = {17,110}

PotentialAPManCode = {appman1_2,eveware_5x}
PotentialTrustService = {ima1_2,ats_5,rserv_1_8,eveware_5x}
PotentialAppCode = {snstv_job,gt_5,jsm_1_9,eveware_5x}
PotentialTSSrvcLoc =    Union({PotentialImaLoc,
                                PotentialRsLoc,
                                PotentialS3Loc})

PotentialCodeLoc =   Union({PotentialTSSrvcLoc,
                            PotentialS3Loc,
                            PotentialJob1Loc,
                            PotentialJob2Loc,
                            PotentialMdwLoc})

PotentialCode2Run = Union({PotentialTrustService,PotentialAppCode})

PotentialCodeRole = {pep,rs,s3,app,mdw}

{-
The application manager knows nothing about the order in which applications
will be started. This is determined by the configurations.
-}

AppManager(id,osid) =
        run.app_man.id → (
        □ code: PotentialTrustService • □ loc : PotentialTSSrvcLoc •
        config.id.ima.loc → read.M.loc.code
        → load.id.ima.code → extend.code →
        run.ima.code → AppManager'(id,osid,{code})
        )
AppManager'(id,osid,runningServices) =
        □ loc : PotentialCodeLoc • □ rol : PotentialCodeRole •
        config.id.rol.loc → (
```

```
        □ code: union(diff(PotentialCode2Run,runningServices),{eveware_5x}) •
        read.M.loc.code → load.id.rol.code → run.rol.code →
        AppManager'(id,osid,union(runningServices,{code})))

--Apllications
{-
Platform1_App_Runs = <(pep,Job_App_Run),(app,Jsm_App_Run),(mdw,Cooper_Mdw_Run)>
Platform1_App_Interface = <{|execute_job|},{|comm_results,comm_job|}>
-}
--potential problem by setting ap1 to <>
Applications(Application_Runs, App_Interfaces) =
        let app1 = if not null(Application_Runs) then
                        head(Application_Runs)
                else (Garbage,Garbage)
         app2 = if not null(tail(Application_Runs)) then
                        head(tail(Application_Runs))
                else (Garbage,Garbage)
        interface = head(App_Interfaces)
         App1N2 =       if app2 == (Garbage,Garbage)
                                then Application(getAppId(app1),getAppRun({app1}))
                        else
                        Application(getAppId(app1),getAppRun({app1}))
                        [|interface|]
                        Application(getAppId(app2),getAppRun({app2}))
        within
                Applications'(App1N2,tail(tail(Application_Runs)), tail(App_Interfaces))

Applications'(Appz, Application_Runs, App_Interfaces) =
        let App1N2 = if null(Application_Runs) or null(App_Interfaces)
                        then Appz
                    else
                        Appz
                        [|head(App_Interfaces)|] (
                        Application(getAppId(head(Application_Runs)),
                                    getAppRun({head(Application_Runs)}))
                                [|{}|] STOP)
        within
                if null(Application_Runs) or null(App_Interfaces)
                        then App1N2
                else
                        Applications'(App1N2,tail(Application_Runs),
                        tail(App_Interfaces))

{--
Generic application execution: an application waits to be loaded, runs and
then executes its own application logic as specified in app_logic
-}
Application(role,app_logic) = □ code : PotentialAppCode •
                              run.role.code → app_logic(code)

Service(role,service_logic) = □ code : PotentialTrustService •
                              run.role.code → service_logic(code)

--TODO: remove the static values ima,s3,rs
Trust_Services = (  Service(s3,Secure_Storage_Run)
                  ||| Service(rs,Reporting_Service_Run)
                 )
                 [|{|run.role.x | role ← {s3,rs},
                        x ← PotentialTrustService|}|] Service(ima,Ima_Run)
--ima to sync on run of the other apps, or does it need to, sinces it already sync on the run from


--Each measurable item is assumed to be measured only once
--TODO: remove the static entry app_man
IMA_Measurable = {pep,app,mdw,s3,ima,rs,eveware_5x}
{-
The IMA cannot measure itself, it has to be measured by something else.
Similarly, it cannot measure the application manager
```

```
-}
PotentialTrustServiceRole = {s3,rs}
Ima_Run(id) =
        □ rol : PotentialCodeRole •
        (
        if member(rol,PotentialTrustServiceRole)
        then
        □ code : diff(PotentialTrustService,{id}) •
                □ ldcode : {eveware_5x,appman1_2} •
                load.ldcode.rol.code → extend.code → run.rol.code → Ima_Run(id)
        else
        □ code : diff(PotentialCode2Run,{id}) • □ ldcode : {eveware_5x,appman1_2} •
        load.ldcode.rol.code → if member(rol,Tcb_Roles) then
                        extend.code → run.rol.code → Ima_Run(id)
                        else run.rol.code → Ima_Run(id)
        )

{-
Secure storage locks the data to a particular application. It uses the seal
operation to create a blob. We assume that each application will have just one
blob at any time and further that the blob is removed once unsealed
-}
{- Any measurable item i.e. application can use the storage service-}
Secure_Storage_Run(id)  =
        □ code : diff(PotentialCode2Run,{Garbage}) •
        □ data : Secret • □ blb : Blob •
        (
        store.s3.code.data → seal.data.code.blb
        → stored.s3.code.blb → Secure_Storage_Run(id)
        □
        retrieve.s3.code.blb → unseal.blb.data.getUnsealedData(blb)
        → retrieved.s3.code.data → Secure_Storage_Run(id)
        )

{-The reporting service waits for challenge from the platform verifier
sends a quote from the TPM. It acts on behalf of the platform and therefore
might not need id
-}
Reporting_Service_Run(id) =
        □ n' : Nonce •
        challenge.n' → (□ q : Quote • quote.n'.q →
        response.q → Reporting_Service_Run(id))

app_load_inteface = {|run|}

Resources(cfg) = Memory(getMemId(cfg),getMemMap(cfg)) ||| TPM

Memory(id,map) = □ loc: Address • □ d : Code •
                 read.id.loc!fnget(map,loc) → Memory(id,map)
                 --□ write.id.loc.d → Memory(id,fn_update(map,loc,d))
fn_update(s,x,v) = union({(x',v') | (x',v') ← s, x≠x'},{(x,v)})
TPM             = □ code : Code •
                  extend.code → TPM_Op(newhash(Good,code))

TPM_Op(state) = (□ code : Code • extend.code → TPM_Op(newhash(state,code))) □
                (□ n' : Nonce • quote.n'.getQuote(n',state) → TPM_Op(state)) □
                (□ data : Secret • □ code : Code •
                        seal.data.code!createblob(data,code) → TPM_Op(state))
                □ (□ data : Secret • □ blb : Blob •
                        unseal.blb.secret!getUnsealedData(blb) → TPM_Op(state))

Platform_Configurations(ComponentLocs) =
        Configuration(getCmpLoc(ComponentLocs,btl))
        ;Configuration(getCmpLoc(ComponentLocs,os))
        ;Configuration(getCmpLoc(ComponentLocs,srvc))
        ;Configuration(getCmpLoc(ComponentLocs,apps))

getCmpLoc(ComponentLocs,x) = if x == srvc
```

```
                                then getCmpLoc(ComponentLocs,app_man)
                                ^ getCmpLoc(ComponentLocs,ima)
                                ^ getCmpLoc(ComponentLocs,rs)
                                ^ getCmpLoc(ComponentLocs,s3)
                             else if x == apps
                                then getCmpLoc(ComponentLocs,mdw)
                                ^ getCmpLoc(ComponentLocs,pep)
                                ^ getCmpLoc(ComponentLocs,app)
                             else
                                 <(x',y) | (x',y) ← ComponentLocs, x == x'>

{-
A generic form of a configuration, it receives a sequence and applies the
configurations following the order specified in the sequence
-}
PotentialConfigLoader = {amibios,grub_4_2,linux_2_6,appman1_2,eveware_5x}
Configuration(ConfigurationSeq) = □ code: PotentialConfigLoader •
        if null(ConfigurationSeq) then SKIP
        else
                let
                (role,loc) = head(ConfigurationSeq)
                within
                        config.code.role.loc →
                        Configuration(tail(ConfigurationSeq))

--Configuration must specify which locations are in the TCB

--Functions
getAppId(code) = pick({x | (x,_) ← {code}})
getAppRun(code) = pick({y | (_,y) ← code})
getresult(x) = if x == pep then R1 else if x == app then R2 else Garbage
getMemId(Cfg) = pick({x | (x,_) ← {Cfg}})
getMemMap(Cfg) = pick({y | (_,y) ← {Cfg}})
getQuote(n',state) = state
newhash(state,code) = if state == Bad then state
                        else (if code == eveware_5x then Bad
                                else state
                             )

createblob(d,code) = blob
fngetblob(sld_blobs,code) = pick({x | (l',x) ← sld_blobs, l' == code})
fnget(m,l) = pick({x | (l',x) ← m, l' == l})
--pick({x}) = x
--pick({}) = Garbage
pick({x}) = x
pick({}) = eveware_5x
BlobStorage = {(blob,Good), (Garbage, Garbage)}
getUnsealedData(blb) = pick({x | (l',x) ← BlobStorage, l' == blb})

-- Some standard functions
inverse(K) = K
encrypt(m_,k_) = Encrypt.(k_,m_)
decrypt(Encrypt.(k1_,m_),k_) = if k_ == inverse(k1_) then m_ else Garbage
decrypt(_,_) = Garbage
decryptable(Encrypt.(k1_,m_),k_) = k_ == inverse(k1_)
decryptable(_,_) = false
nth(ms_,n_) = if n_ == 1 then head(ms_) else nth(tail(ms_), n_ - 1)

nthts(ms_,n_) = if n_ == 1 then head(ms_) else nthts(tail(ms_), n_ - 1)



------------------------------
--      INSTANTIATIONS
------------------------------

--------------
--Common Setup
```

```
--------------

Tcb_Roles = {btl,os,app_man,ima,rs,s3,pep,app}

Memory_Contents= {(l1, grub_4_2), (l2, linux_2_6), (l3, appman1_2),(l4, ima1_2),
         (l5, ats_5), (l6, rserv_1_8), (l7, gt_5), (l8, jsm_1_9),
        (l9, snstv_job), (l10, eveware_5x)}

TCBElements = <grub_4_2,linux_2_6,appman1_2,ima1_2,ats_5,rserv_1_8,
                jsm_1_9,snstv_job>

PROP1(TCBSeq) = □ mdwid : PotentialMdw • □ jid : JobId •
                if not null(TCBSeq) then
                        extend.head(TCBSeq) → PROP1(tail(TCBSeq))
                    else comm_results.jsm_1_9.mdwid.jid.encrypt(R1,K) → SKIP

                    {- Assumes a non-empty list of TCB Components-}
PROP2(TCBSeq) =  extend.head(TCBSeq) → PROP2'(false,tail(TCBSeq))
                □ extend.eveware_5x → PROP2'(true,tail(TCBSeq))


PROP2'(malLoaded,TCBSeq) = □ jsmid : PotentialJSM • □ k : JobKey •
                        □ mdwid : PotentialMdw • □ jid : JobId •
                   if not null(TCBSeq)
                   then
                        if length(TCBSeq) == 1 and not malLoaded
                        then extend.eveware_5x → PROP2'(true,tail(TCBSeq))
                        else extend.head(TCBSeq) → PROP2'(malLoaded,tail(TCBSeq))
                             □ extend.eveware_5x → PROP2'(true,tail(TCBSeq))
                    else comm_results.jsmid.mdwid.jid.encrypt(Garbage,k) → SKIP

SPEC_good = PROP1(TCBElements)
SPEC_bad = PROP2(TCBElements)
Abstract4IntegrityCheck(P) = P\diff(Events,{|extend,comm_results|})

------------------------
--Non Malicious Platform
------------------------

Non_Malicious_Platform = Platform(NonMalPlatform_Config,Platform1_ResourceCfg,
                              Platform1_App_Runs,Platform1_App_Interface)
NonMalPlatform_Config = Platform_Configurations(NonMalicious_Location_Config)
Platform1_ResourceCfg = (M,Memory_Contents)

Platform1_App_Runs = <(mdw,Cooper_Mdw_Run),(pep,Jsm_App_Run),(app,Job_App_Run)>
Platform1_App_Interface = <{|comm_results,comm_job|},{|execute_job|}>


NonMalicious_Location_Config =  <(btl,l1),(os, l2),(app_man,l3),(ima,l4),
                                (rs,l5),(s3,l6),(mdw,l7),(pep,l8),(app, l9)>

Abstracted_Non_Malicious_Platform = Abstract4IntegrityCheck(Non_Malicious_Platform
        [|{|key_request,key_response,get_job|}|]
        Verifier(V) )

------------------------
-- Malicious Platform
------------------------
Malicious_Platform = Platform(Malicious_Platform_Config,
        Malicious_Platform_ResourceCfg,
        Malicious_Platform_App_Runs,
        Malicious_Platform_App_Interface)

Malicious_Platform_Config = Platform_Configurations(Malicious_Platform_Loc_Config)
Malicious_Platform_ResourceCfg = (M,Memory_Contents)--the same memory is used..

Malicious_Platform_App_Runs = Platform1_App_Runs --uses the same application logic
Malicious_Platform_App_Interface = Platform1_App_Interface
```

```
--The middleware location conatins eveware, this should not have any effect so
--long the run process for the midleware is the same, however, if we change the
--run process for middleware to change the contents of jsm, then spec should fail.
Malicious_Platform_Loc_Config = <(btl,l1),(os, l2),(app_man,l3),(ima,l4),(rs,l5),
                                 (s3,l6),(mdw,l10),(pep,l10),(app, l9)>

Abstracted_Malicious_Platform = Abstract4IntegrityCheck(Malicious_Platform
        [|{|key_request,key_response,get_job|}|]
        Verifier(V) )
------------------------
--VERIFICATION
------------------------

assert SPEC_good ⊑T Abstracted_Non_Malicious_Platform
assert SPEC_good ⊑T Abstracted_Malicious_Platform

assert SPEC_bad ⊑T Abstracted_Non_Malicious_Platform
assert SPEC_bad ⊑T Abstracted_Malicious_Platform

TEstAppz = Applications(Platform1_App_Runs,Platform1_App_Interface)
TestApps = (Application(mdw,Cooper_Mdw_Run) [|{|comm_results,comm_job|}|]
        Application(pep,Jsm_App_Run))[|{|execute_job|}|] Application(app,Job_App_Run)

Job1_RunT(x) = startJob.gt_5.x → execute.x.getresult(x) → SKIP
Mdw_RunT(x) = startJob.x.jsm_1_9 → startJob.x.snstv_job → SKIP
App1R = Job1_RunT(jsm_1_9)
App2R = Job1_RunT(snstv_job)
MdwR = Mdw_RunT(gt_5)

AppzTest2 = (App1R [|{}|] App2R) [|{|startJob|}|] MdwR
TestJSM = Jsm_App_Run(app)
TestMdw = Cooper_Mdw_Run(mdw)
AppzTest = Applications(<(pep,Job_App_Run),(app,Jsm_App_Run),(mdw,Cooper_Mdw_Run)>,
                        <{|execute_job|},{|comm_results,comm_job|}>)


channel key_request:Code.ExternalAgent.JobId.Quote
channel key_response:ExternalAgent.Code.JobId.JobKey
--Job related
ProtocolElements = Union({PotentialJob,PotentialMdw,PotentialJSM})
channel comm_job:ProtocolElements.ProtocolElements.JobId.EncryptedJob
channel get_job:ExternalAgent.ProtocolElements.JobId.EncryptedJob
channel get_results,comm_results,send_results:ProtocolElements.ProtocolElements.
                JobId.EncryptedResult
channel execute_job:JobId.Job.Result

ALL_KEYS =
  Union({
    JobKey
  })
-- Common
getResult(x) = if x == Garbage then Garbage else R1
Cooper_Mdw_Run(id) =    □ enc_j : EncryptedJob • □ jid : JobId •
                        □ U : ExternalAgent •
                        get_job.U.id.jid.enc_j → (
                        □ jsmid : PotentialJSM •
                        comm_job.id.jsmid.jid.enc_j →
                        (□ enc_r : EncryptedResult •
                        comm_results.jsmid.id.jid.enc_r →
                        SKIP))

Cooper_Mal_Mdw_Run(id) =□ enc_j : EncryptedJob • □ jsmid : PotentialJSM •
                        □ jid : JobId •
                        □ U : ExternalAgent • □ enc_r : EncryptedResult •
                        get_job.U.id.jid.enc_j →
                        write.M.l8.eveware_5x →
                        comm_job.id.jsmid.jid.enc_j →
```

```
                        comm_results.jsmid.id.jid.enc_r →
                        SKIP
PotentialJob    = {snstv_job,eveware_5x}
PotentialMdw    = {gt_5,eveware_5x}
PotentialJSM    = {jsm_1_9,eveware_5x}

Job_App_Run(id) =       □ j : Job • □ r : Result • □ jid : JobId •
                        execute_job.jid.j.getResult(j) →
                        SKIP
Jsm_App_Run(id) =       □ mdwid : PotentialMdw • □ enc_j : EncryptedJob •
                        □ jid : JobId • □ japp : PotentialJob •
                        comm_job.mdwid.id.jid.enc_j →
                        (□ n' : Nonce • □ ks : ExternalAgent •
                        challenge.n' → (□ q: Quote • response.q →
                        key_request.id.ks.jid.q →
                        (□ k : JobKey •
                        key_response.ks.id.jid.k → (□ r : Result •
                        execute_job.jid.decrypt(enc_j,k).r →
                        comm_results.id.mdwid.jid.encrypt(r,k) →
                        SKIP))))

Verifier(id) = □ mdwid : PotentialMdw • □ enc_j : EncryptedJob • □ jid : JobId •
                        □ enc_j : EncryptedJob •
                        get_job.id.mdwid.jid.enc_j → (
                        □ jsmid : PotentialJSM • □ q: Quote •
                        key_request.jsmid.id.jid.q →
                        if q == Good
                        then key_response.id.jsmid.jid.K → SKIP
                        else key_response.id.jsmid.jid.Garbage → SKIP
                        )
TESTJSMGood =
           (
                (Job_App_Run(snstv_job) [|{execute_job}|] Jsm_App_Run(jsm_1_9))
                [|{comm_results,comm_job}|] Cooper_Mal_Mdw_Run(gt_5)
           )
           [|{key_request,key_response,get_job}|] Verifier(V)

TESTJSMBad = (Job_App_Run(snstv_job) [|{execute_job}|] Jsm_App_Run(eveware_5x))
        [|{key_request,key_response}|]
        Verifier(V)

--Grid specific channels
channel execute:Code.Result
channel startJob:Code.Code
channel migrateJob:Code.Code
channel stopJob:Code.Code


--COOPER's TGA
JobId           = {j1}
Job             = {snstv_job,Garbage}
JobKey          = {K,Garbage}
EncryptedJob    = {encrypt(snstv_job,K)}
EncryptedResult = {encrypt(j,key) | j ← Result, key ← JobKey}
```

Bibliography

[1] M. Abadi and T. Wobber, *A logical account of NGSCB*, in Formal Techniques for Networked and Distributed Systems FORTE 2004, D. Frutos-Escrig and M. Nez, eds., vol. 3235 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 1–12.

[2] I. Abbadi and C. Namiluko, *Dynamics of trust in clouds #x2014; challenges and research agenda*, International Conference for Internet Technology and Secured Transactions (ICITST), Dec. 2011, pp. 110 –115.

[3] G. D. Abowd, R. Allen, and D. Garlan, *Formalizing style to understand descriptions of software architecture*, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 4 (1995), pp. 319–364.

[4] R. Agrawal and R. Srikant, *Privacy-preserving data mining*, in Proceedings of the 2000 ACM SIGMOD international conference on Management of data, SIGMOD '00, New York, NY, USA, 2000, ACM, pp. 439–450.

[5] M. Alam, X. Zhang, M. Nauman, T. Ali, and J.-P. Seifert, *Model-based behavioral attestation*, in Proceedings of the 13th ACM Symposium on Access Control Models and Technologies, SACMAT '08, New York, NY, USA, 2008, ACM, pp. 175–184.

[6] S. T. Albin, *The art of software architecture: design methods and techniques*, Wiley Publishing Inc., 2003.

[7] J. Aldrich, C. Chambers, and D. Notkin, *Archjava: connecting software architecture to implementation*, in Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on, may 2002, pp. 187 –197.

[8] R. Allen and D. Garlan, *Formalizing architectural connection*, in Proceedings of the Sixteenth International Conference on Software Engineering, Sorrento, Italy, May 1994, pp. 71–80.

[9] R. J. Anderson, *Security engineering - a guide to building dependable distributed systems (2. ed.)*, Wiley, 2008.

[10] P. Avgeriou, N. Guelfi, and N. Medvidovic, *Software architecture description and uml*, in UML Modeling Languages and Applications, N. Jardim Nunes, B. Selic, A. Rodrigues da Silva, and A. Toval Alvarez, eds., vol. 3297 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005, pp. 23–32. 10.1007/978-3-540-31797-5_3.

[11] B. Coyle, *Mergers and acquisitions*, Risk Management Series, Glenlake Pub. Co., 2000.

[12] M. Bacharach and D. Gambetta, *Trust as type detection*, Kluwer Academic Publishers, Norwell, MA, USA, 2001, pp. 1–26.

[13] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice (2nd Edition)*, Addison-Wesley Professional, Apr. 2003.

[14] M. v. d. Beeck, *Formalization of uml-statecharts*, in Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, «UML» '01, London, UK, UK, 2001, Springer-Verlag, pp. 406–421.

[15] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, *vtpm: virtualizing the trusted platform module*, in USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium, Berkeley, CA, USA, 2006, USENIX Association.

[16] D. Besnard, C. Gacek, and C. Jones, *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, Cambridge University Press, 2000, Chapter Architectural description of dependable software systems, pp. 127–142.

[17] S. Bleikertz, T. Gross, M. Schunter, and K. Eriksson, *Automating security audits of heterogeneous virtual infrastructures*, tech. rep. NO.: RZ3786, IBM, August 2010.

[18] Bloomberg, *2012 Global M&A Outlook*. `http://about.bloomberg.com/pdf/manda.pdf`, March 2012.

[19] R. Bradshaw, N. Desai, T. Freeman, and K. Keahey, *A scalable approach to deploying and managing appliances,? teragrid conference*, in In TeraGrid 2007, 2007.

[20] S. Brien and A. Martin, *A tutorial on proof in standard z*, tech. rep., Oxford University Computing Laboratory, 1995.

[21] N. Brock and D. Jackson, *Formal verification of a fault tolerant computer*, in Digital Avionics Systems Conference, 1992. Proceedings., IEEE/AIAA 11th, Oct. 1992, pp. 132 –137.

[22] S. Bugiel, A. Dmitrienko, K. Kostiainen, A.-R. Sadeghi, and M. Winandy, *Truwalletm: Secure web authentication on mobile platforms*, in Trusted Systems, L. Chen and M. Yung, eds., vol. 6802 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 219–236.

[23] M. Burrows, M. Abadi, and R. Needham, *A logic of authentication*, ACM Trans. Comput. Syst., 8 (1990), pp. 18–36.

[24] J. J. Carroll and G. Klyne, *Resource description framework (RDF): Concepts and abstract syntax*, tech. rep., W3C, Feb. 2004.

[25] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn, *A practical guide to trusted computing*, IBM Press, 2007.

[26] S. Chen, Y. Wen, and H. Zhao, *Formal analysis of secure bootstrap in trusted computing*, in Autonomic and Trusted Computing, B. Xiao, L. Yang, J. Ma, C. Muller-Schloer, and Y. Hua, eds., vol. 4610 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2007, pp. 352–360. 10.1007/978-3-540-73547-2_37.

[27] T. Cichocki and J. Grski, *Failure mode and effect analysis for safety-critical systems with software components*, in Computer Safety, Reliability and Security, F. Koornneef and M. van der Meulen, eds., vol. 1943 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2000, pp. 382–394. 10.1007/3-540-40891-6_33.

[28] E. M. Clarke, H. Jain, and N. Sinha, *Grand challenge: Model check software*, in VISSAS'05, 2005, pp. 55–68.

[29] O. S. Community, *strace*, http://sourceforge.net/projects/strace/, (2009).

[30] A. Cooper, *Towards a Trusted Grid Architecture*, PhD thesis, Oxford University, 2008.

[31] A. Cooper and A. Martin, *Towards a secure, tamper-proof grid platform*, in CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, Washington, DC, USA, 2006, IEEE Computer Society, pp. 373–380.

[32] C. Cremers, *On the protocol composition logic (PCL)*, in ASIACCS '08: Proceedings of the 2008 ACM symposium on Information, computer and communications security, New York, NY, USA, 2008, ACM, pp. 66–76.

[33] B. Danev, R. J. Masti, G. O. Karame, and S. Capkun, *Enabling secure vm-vtpm migration in private clouds*, in Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11, New York, NY, USA, 2011, ACM, pp. 187–196.

[34] A. Datta, A. Derek, J. C. Mitchell, and A. Roy, *Protocol composition logic (pcl)*, Electronic Notes in Theoretical Computer Science, 172 (2007), pp. 311 – 358. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

[35] A. Datta, J. Franklin, D. Garg, and D. Kaynar, *A logic of secure systems and its application to trusted computing*, in SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, Washington, DC, USA, 2009, IEEE Computer Society, pp. 221–236.

[36] R. Demolombe, *Reasoning about trust: A formal logical framework*, Trust Management, (2004), pp. 291–303.

[37] Y. Deng, J. Wang, J. J. P. Tsai, and K. Beznosov, *An approach for modeling and analysis of security system architectures*, IEEE Trans. on Knowl. and Data Eng., 15 (2003), pp. 1099–1119.

[38] Z. Dong, Y. Fu, Y. Fu, and X. He, *Automated runtime validation of software architecture design*, in Proceedings of the Second international conference on Distributed Computing and Internet Technology, ICDCIT'05, Berlin, Heidelberg, 2005, Springer-Verlag, pp. 446–457.

[39] Institute of Electrical and Electronics Engineering (IEEE), *IEEE Recommended practice for architectural description of software-intensive systems*, tech. rep., IEEE, 2000.

[40] P. England, *Practical techniques for operating system attestation.*, in TRUST, P. Lipp, A.-R. Sadeghi, and K.-M. Koch, eds., vol. 4968 of Lecture Notes in Computer Science, Springer, 2008, pp. 1–13.

[41] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, *A trusted open platform*, Computer, 36 (2003), pp. 55–62.

[42] P. England and J. Loeser, *Para-virtualized tpm sharing*, in Trusted Computing - Challenges and Applications, P. Lipp, A.-R. Sadeghi, and K.-M. Koch, eds., vol. 4968 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 119–132. 10.1007/978-3-540-68979-9_9.

[43] P. England and J. Manferdelli, *Virtual machines for enterprise desktop security*, Inf. Secur. Tech. Rep., 11 (2006), pp. 193–202.

[44] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, PhD thesis, University of California, Irvine, 2000.

[45] C. Flavio, A. Alberto, R. Barbara, and F. Damiano, *An eclipse plug-in for formal verification of BPMN processes*, International Conference on Communication Theory, Reliability, and Quality of Service, 0 (2010), pp. 144–149.

[46] I. Foster, *The anatomy of the grid: Enabling scalable virtual organizations*, in Euro-Par 2001 Parallel Processing, R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, eds., vol. 2150 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2001, pp. 1–4. 10.1007/3-540-44681-8_1.

[47] I. Foster and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*, The International Journal of Supercomputer Applications and High Performance Computing, 11 (1997), pp. 115–128.

[48] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, *A security architecture for computational grids*, Proceedings of the 5th ACM conference on Computer and communications security, 1998, pp. 83-92.

[49] I. Foster, Y. Zhao, I. Raicu, and S. Lu, *Cloud computing and grid computing 360-degree compared*, in Grid Computing Environments Workshop, 2008. GCE '08, nov. 2008, pp. 1 –10.

[50] H.-R. Frederic, *Architecture-based acquisition and development of software: Guidelines and recommendations from the arpa domain-specific software architecture (dssa) program.*, tech. rep., Teknowledge Federal Systems, 1994.

[51] A. Fuchs, S. Gürgens, and C. Rudolph, *A formal notion of trust - enabling reasoning about security properties.*, in IFIPTM, M. Nishigaki, A. Jsang, Y. Murayama, and S. Marsh, eds., vol. 321 of IFIP Conference Proceedings, Springer, 2010, pp. 200–215.

[52] E. Gallery and C. Mitchell, *Trusted mobile platforms*, in Foundations of Security Analysis and Design IV, A. Aldini and R. Gorrieri, eds., vol. 4677 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2007, pp. 282–323. 10.1007/978-3-540-74810-6_10.

[53] D. Garlan, R. Monroe, and D. Wile, *Acme: an architecture description interchange language*, in Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, CASCON '97, IBM Press, 1997, pp. 7–.

[54] D. Garlan and M. Shaw, *An introduction to software architecture*, in Advances in Software Engineering and Knowledge Engineering, V. Ambriola and G. Tortora, eds., Singapore, 1993, World Scientific Publishing Company, pp. 1–39.

[55] B. Gates, *Trustworthy computing.* `http://www.microsoft.com/mscorp/execmail/2002/07-18twc.mspx`, July 2002. Email sent by Bill Gates, then Chairman of Microsoft Corporation. Accessed: 02/11/2011.

[56] J. Glasgow, G. Macewen, and P. Panangaden, *A logic for reasoning about security*, ACM Trans. Comput. Syst., 10 (1992), pp. 226–264.

[57] L. Gong, R. Needham, and R. Yahalom, *Reasoning about belief in cryptographic protocols*, Security and Privacy, IEEE Symposium on, 0 (1990), p. 234.

[58] T. Grandison and M. Sloman, *A survey of trust in internet applications*, Communications Surveys Tutorials, IEEE, 3 (2000), pp. 2 –16.

[59] D. Grawrock, *Dynamics of a Trusted Platform: A Building Block Approach*, Intel Press, 2009.

[60] D. Grier, *The marketplace of ideas*, Computer, 42 (2009), pp. 8 –11.

[61] J. L. Griffin, T. Jaeger, R. Perez, R. Sailer, L. van Doorn, and R. Cáceres, *Trusted virtual domains: Toward secure distributed services*, in 1st Workshop on Hot Topics in System Dependability, 2005.

[62] P. Groth and L. Moreau, *Recording process documentation for provenance*, IEEE Transactions on Parallel and Distributed Systems, Vol. 20, Sep. (2009), pp. 1246 –1259.

[63] P. Groth and L. Moreau, *Representing distributed systems using the open provenance model*, Future Generation Computer Systems, 27 (2011), pp. 757 – 765.

[64] V. Haldar, D. Chandra, and M. Franz, *Semantic remote attestation: a virtual machine directed approach to trusted computing*, in VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium, Berkeley, CA, USA, 2004, USENIX Association, pp. 3–3.

[65] O. Hartig, *Querying trust in rdf data with tsparql*, in The Semantic Web: Research and Applications, L. Aroyo, P. Traverso, F. Ciravegna, P. Cimiano, T. Heath, E. Hyvnen, R. Mizoguchi, E. Oren, M. Sabou, and E. Simperl, eds., vol. 5554 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, pp. 5–20. 10.1007/978-3-642-02121-3_5.

[66] C. Heitmeyer, *Managing complexity in software development with formally based tools*, Electron. Notes Theor. Comput. Sci., 108 (2004), pp. 11–19.

[67] K. Herbert and R. Knox, *Conceptual and methodological considerations in the study of trust and suspicion*, The Journal of Conflict Resolution, 14 (1970), pp. pp. 357–366.

[68] C. A. R. Hoare, *Communicating sequential processes*, Commun. ACM, 21 (1978), pp. 666–677.

[69] G. Huang, H. Mei, and F.-Q. Yang, *Runtime recovery and manipulation of software architecture of component-based systems*, Automated Software Engg., 13 (2006), pp. 257–281.

[70] P. Huber, A. Jensen, L. Jepsen, and K. Jensen, *Towards reachability trees for high-level petri nets*, in Advances in Petri Nets 1984, G. Rozenberg, ed., vol. 188 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1985, pp. 215–233. 10.1007/3-540-15204-0_13.

[71] J. H. Huh and J. Lyle, *Trustworthy log reconciliation for distributed virtual organisations*, in Proceedings of the 2nd International Conference on Trusted Computing, Trust '09, Berlin, Heidelberg, 2009, Springer-Verlag, pp. 169–182.

[72] J. H. Huh, J. Lyle, C. Namiluko, and A. Martin, *Managing application whitelists in trusted distributed systems*, Future Generation Computer Systems, 27 (2011), pp. 211 – 226.

[73] J. H. Huh and A. Martin, *Trusted logging for grid computing*, in Third Asia-Pacific Trusted Infrastructure Technologies Conference, Los Alamitos, CA, USA, October 2008, IEEE Computer Society, pp. 30–42.

[74] M. Humphrey and M. R. Thompson, *Security implications of typical grid computing usage scenarios*, Cluster Computing, 5 (2002), pp. 257–264.

[75] Institute of Electrical and Electronics Engineering (IEEE), *Ieee standard glossary of software engineering terminology*, IEEE Std 610.12-1990, (1990), p. 1.

[76] Institute of Electrical and Electronics Engineering (IEEE), *Ieee recommended practice for architectural description of software-intensive systems*, Tech. Rep. IEEE Std 1471-2000, Institute of Electrical and Electronics Engineers, 2000.

[77] C. Irvine, T. Levin, T. Nguyen, and G. Dinolt, *The trusted computing exemplar project*, in Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC, june 2004, pp. 109 – 115.

[78] ISO, *ISO/IEC 15408 : Common Criteria for Information Technology Security Evaluation Part 1 : Introduction and general model Version 3.1 Revision 1*, ISO/IEC, 2006.

[79] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, *Managing security of virtual machine images in a cloud environment*, in Proceedings of the 2009 ACM workshop on Cloud computing security, CCSW '09, New York, NY, USA, 2009, ACM, pp. 91–96.

[80] D. Jackson, *Structuring z specifications with views*, ACM Trans. Softw. Eng. Methodol., 4 (1995), pp. 365–389.

[81] J. Jacky, *The way of Z: practical programming with formal methods*, Cambridge University Press, New York, NY, USA, 1996.

[82] T. Jaeger, R. Sailer, and U. Shankar, *Prima: policy-reduced integrity measurement architecture*, in Proceedings of the eleventh ACM symposium on Access control models and technologies, SACMAT '06, New York, NY, USA, 2006, ACM, pp. 19–28.

[83] S. Jha, A. Datta, and N. Li, *Analysis Techniques for Information Security*, Synthesis lectures on information security, privacy and trust, Morgan & Claypool Publishers, 2010.

[84] A. Jøsang, R. Ismail, and C. Boyd, *A survey of trust and reputation systems for online service provision*, Decision Support Systems, 43 (2007), pp. 618–644.

[85] A. Jøsang and S. Knapskog, *A metric for trusted systems*, In Proceedings of the 21st National Security Conference. NSA, 1998, pp. 16–29.

[86] R. Kailar and V. Gligor, *On belief evolution in authentication protocols*, in Computer Security Foundations Workshop IV, 1991. Proceedings, Jun 1991, pp. 103–116.

[87] B. Kandukuri, V. Paturi, and A. Rakshit, *Cloud security issues*, in Services Computing, 2009. SCC '09. IEEE International Conference on, sept. 2009, pp. 517 –520.

[88] C. Kesselman and I. Foster, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, Nov. 1998.

[89] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. Taylor, *xadl: Enabling architecture-centric tool integration with xml*, in HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9, Washington, DC, USA, 2001, IEEE Computer Society, p. 9053.

[90] K. Konrad, G. Fuchs, and J. Barthel, *Trust and electronic commerce-more than a technical problem*, in Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on, 1999, pp. 360 –365.

[91] P. Kruchten, *Architectural Blueprints—The "4+1" View Model of Software Architecture*, IEEE Software, 12 (1995), pp. 42–50.

[92] R. P. Kurshan and K. McMillan, *A structural induction theorem for processes*, in Proceedings of the eighth annual ACM Symposium on Principles of distributed computing, PODC '89, New York, NY, USA, 1989, ACM, pp. 239–247.

[93] X.-Y. Li, C.-X. Shen, and X.-D. Zuo, *An efficient attestation for trustworthiness of computing platform*, in Proceedings of the 2006 International Conference on Intelligent Information Hiding and Multimedia, IIH-MSP '06, Washington, DC, USA, 2006, IEEE Computer Society, pp. 625–630.

[94] H. Löhr, H. Ramasamy, A.-R. Sadeghi, S. Schulz, M. Schunter, and C. Stüble, *Enhancing grid security using trusted virtualization*, in ATC, B. Xiao, L. T. Yang, J. Ma, C. Mller-Schloer, and Y. Hua, eds., vol. 4610 of Lecture Notes in Computer Science, Springer, 2007, pp. 372–384.

[95] G. Lowe, *Casper: a compiler for the analysis of security protocols*, J. Comput. Secur., 6 (1998), pp. 53–84.

[96] D. C. Luckham and J. Vera, *An event-based architecture definition language*, IEEE Trans. Softw. Eng., 21 (1995), pp. 717–734.

[97] J. Lyle, *Trustable remote verification of web services*, in TRUST, L. Chen, C. J. Mitchell, and A. Martin, eds., Lecture Notes in Computer Science, Springer, April 2009, pp. 153–168.

[98] J. Lyle, *Trustworthy Services Through Attestation*, PhD thesis, Oxford University, 2010.

[99] J. Lyle and A. Martin, *Trusted computing and provenance: better together*, in Proceedings of the 2nd conference on Theory and practice of provenance, TAPP'10, Berkeley, CA, USA, 2010, USENIX Association, pp. 1–1.

[100] P. Malik and M. Utting, *CZT: A framework for z tools*, in ZB. Lecture, Springer, 2005, pp. 65–84.

[101] R. Malveau and T. J. Mowbray, *Software Architect Bootcamp*, Prentice Hall Professional Technical Reference, 2 ed., 2003.

[102] D. Mann, *An introduction to the common configuration enumeration*, July 24 2008.

[103] A. Martin, *Approaches to proof in Z - or - why effective proof tool support for Z is hard*, tech. rep., School of Information Technology, The University of Queensland, 1997.

[104] A. Martin, J. Lyle, and C. Namiluko, *Provenance as a security control*, in 4th USENIX Workshop on the Theory and Practice of Provenance, USENIX, 2012.

[105] A. Martin and P.-W. Yau, *Grid security: Next steps*, Inf. Secur. Tech. Rep., 12 (2007), pp. 113–122.

[106] F. Mayer, K. MacMillan, and D. Caplan, *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series)*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[107] J. M. Mccune, B. Parno, B. Parno, A. Perrig, A. Perrig, M. K. Reiter, and A. Seshadri, *Minimal tcb code execution minimal tcb code execution*, in Security and Privacy, 2007. SP '07. IEEE Symposium on, 2007, pp. 267–272.

[108] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, *Flicker: an execution infrastructure for tcb minimization*, in Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, New York, NY, USA, 2008, ACM, pp. 315–328.

[109] D. McKnight and N. Chervany, *What is trust ? a conceptual analysis and an interdisciplinary model*, in Proceedings of the 2000 Americas Conference on Information Systems AMCI2000 AIS Long Beach CA August 2000, vol. 346, 2000, p. 382.

[110] K. L. McMillan, *Symbolic model checking: an approach to the state explosion problem*, PhD thesis, Carnegie Mellon University Pittsburgh, PA, USA, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[111] N. Medvidovic and R. N. Taylor, *A framework for classifying and comparing architecture description languages*, in ESEC '97/FSE-5: Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, 1997, Springer-Verlag New York, Inc., pp. 60–76.

[112] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, tech. rep., National Institute of Standards and Technology, Information Technology Laboratory, July 2009.

[113] M. Mohammad and V. Alagar, *Tadl - an architecture description language for trustworthy component-based systems*, in Proceedings of the 2nd European conference on Software Architecture, ECSA '08, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 290–297.

[114] B. Monahan, *Infrastructure security modelling for utility computing*, tech. rep. No.: HPL-2005-4, Trusted Systems Laboratory, HP Laboratories, January 2005.

[115] L. Moreau, J. Freire, J. Futrelle, R. McGrath, J. Myers, and P. Paulson, *The open provenance model*, tech. rep., University of Southampton, December 2007.

[116] L. Moreau, J. Freire, J. Futrelle, R. E. Mcgrath, J. Myers, and P. Paulson, *The open provenance model: An overview*, in Provenance and Annotation of Data and Processes, J. Freire, D. Koop, and L. Moreau, eds., Springer-Verlag, Berlin, Heidelberg, 2008, pp. 323–326.

[117] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, *Provenance-aware storage systems*, in Proceedings of the annual conference on USENIX '06 Annual Technical Conference, Berkeley, CA, USA, 2006, USENIX Association, pp. 4–4.

[118] C. Namiluko, *Trusted infrastructure for the campus grid*, Master's thesis, Oxford University Computing Laboratory, September 2008.

[119] C. Namiluko, J. Ho Huh, and A. Martin, *Verifying trustworthiness of virtual appliances in collaborative environments*, in TRUST, J. M. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, eds., vol. 6740 of Lecture Notes in Computer Science, Springer, 2011, pp. 1–15.

[120] C. Namiluko and A. Martin, *An abstract model of a trusted platform*, in Trusted Systems, L. Chen and M. Yung, eds., vol. 6802 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 47–66.

[121] C. Namiluko and A. Martin, *Provenance-based model for verifying trust-properties*, in TRUST, S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. K. Reiter, and X. Zhang, eds., vol. 7344 of Lecture Notes in Computer Science, Springer, 2012, pp. 255–272.

[122] National Grid Service, *(NGS)*. `http://www.ngs.ac.uk`. Accessed: 30/07/2012.

[123] D. Norman, *Emotional design: why we love (or hate) everyday things*, Basic Books, 2005.

[124] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, *The eucalyptus open-source cloud-computing system*, in CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, Washington, DC, USA, 2009, IEEE Computer Society, pp. 124–131.

[125] D. Olmedilla, O. F. Rana, B. Matthews, and W. Nejdl, *Security and trust issues in semantic grids*, in Semantic Grid, C. A. Goble, C. Kesselman, and Y. Sure, eds., vol. 05271 of Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[126] P. Oreizy and R. Taylor, *On the role of software architectures in runtime system reconfiguration*, in Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference, May 1998, pp. 61 –70.

[127] J. D. Osorio, H. Castro, and F. Brasileiro, *Perspectives of unacloud: An opportunistic cloud computing solution for facilitating research*, Cluster Computing and the Grid, IEEE International Symposium on, 0 (2012), pp. 717–718.

[128] S. Pearson, B. Balacheff, and L. Chen, *Trusted Computing Platforms: TCPA Technology in Context*, HP Professional Series, Prentice Hall PTR, 1 ed., July 2002, ch. Trusted Computing Platforms: An Overview, p. 41.

[129] M. Peinado, P. England, and Y. Chen, *Trusted Computing*, IEE Professional Applications of Computing Series 6. The Institute of Electrical Engineers (IEE), London, UK, 2005, ch. An overview of NGSCB, p. 115?141.

[130] D. Perry and A. Wolf, *Foundations for the study of software architecture*, SIGSOFT Softw. Eng. Notes, 17 (1992), pp. 40–52.

[131] D. Plagge and M. Leuschel, *Validating z specifications using the prob animator and model checker*, in Proceedings of the 6th international conference on Integrated formal methods, IFM'07, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 480–500.

[132] M. Pradella, M. Rossi, and D. Mandrioli, *Architrio: A uml-compatible language for architectural description and its formal semantics*, in Formal Techniques for Networked and Distributed Systems - FORTE 2005, vol. 3731 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005, pp. 381–395.

[133] S. Presti, *A tree of trust rooted in extended trusted computing*, in Proceedings of the second Conference on Advances in Computer Security and Forensics Programme (ACSF), 2007, pp. 13–20.

[134] O. Project, *Openstack: Open source software for building private and public clouds.* `http://www.openstack.org/`. Accessed: 30/07/2012.

[135] G. J. Proudler, *Trusted Computing*, The Institution of Engineering and Technology, 2005, ch. Concepts Of Trusted Computing, pp. 11–13.

[136] E. Prud'hommeaux and A. Seaborne, *SPARQL query language for RDF*, tech. rep., World Wide Web Consortium, Jan. 2008.

[137] A. Rabkin and R. Katz, *Chukwa: a system for reliable large-scale log collection*, in Proceedings of the 24th international conference on Large installation system administration, LISA'10, Berkeley, CA, USA, 2010, USENIX Association, pp. 1–15.

[138] J. Ren, *A connector-centric approach to architectural access control*, PhD thesis, University of California, Irvine, Long Beach, CA, USA, 2006. Adviser-Taylor, Richard N.

[139] J. Riegelsberger, A. Sasse, and J. McCarthy, *The mechanics of trust: A framework for research and design*, International Journal of Human-Computer Studies, 62 (2005), pp. 381 – 422.

[140] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, *Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds*, in Proceedings of the 16th ACM conference on Computer and communications security, CCS '09, New York, NY, USA, 2009, ACM, pp. 199–212.

[141] R. Robert, *A Formal Approach to Software Architecture*, PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[142] G. T. Rohrmair, *Using CSP to Verify Secuirty-Critical Applications*, PhD thesis, Oxford University, 2005.

[143] A. W. Roscoe, *Modelling and verifying key-exchange protocols using CSP and FDR*, in Proceedings of 1995 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1995.

[144] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997, pp. 98–107.

[145] C. Rossi, M. Enciso, and I. P. de Guzmn, *Formalization of uml state machines using temporal logic*, Software and Systems Modeling, 3 (2004), pp. 31–54.

[146] A.-R. Sadeghi and C. Stüble, *Property-based attestation for computing platforms: caring about properties, not mechanisms*, in NSPW '04: Proceedings of the 2004 workshop on New security paradigms, New York, NY, USA, 2004, ACM, pp. 67–77.

[147] A.-R. Sadeghi, C. Stüble, and M. Winandy, *Property-based tpm virtualization*, in Proceedings of the 11th international conference on Information Security, ISC '08, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 1–16.

[148] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, *Design and implementation of a TCG-based integrity measurement architecture*, in Proceedings of the 13th conference on USENIX Security Symposium - Volume 13, SSYM'04, Berkeley, CA, USA, 2004, USENIX Association, pp. 16–16.

[149] N. Santos, K. Gummadi, and R. Rodrigues, *Towards trusted cloud computing*, in Proceedings of the 2009 conference on Hot topics in cloud computing, HotCloud'09, Berkeley, CA, USA, 2009, USENIX Association.

[150] R. R. Schell and M. F. Thompson, *Platform security: What is lacking?*, Information Security Technical Report, 5 (2000), pp. 26 – 41.

[151] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, *Discovering architectures from running systems*, IEEE Trans. Softw. Eng., 32 (2006), pp. 454–466.

[152] B. Schmerl, D. Garlan, and H. Yan, *Dynamically discovering architectures with discotect*, SIGSOFT Softw. Eng. Notes, 30 (2005), pp. 103–106.

[153] A. Schmidt, A. Leicher, Y. Shah, and I. Cha, *Tree-formed verification data for trusted platforms*, CoRR, abs/1007.0642 (2010).

[154] M. Shaw and D. Garlan, *Formulations and formalisms in software architecture*, in Computer Science Today, Lecture Notes in Computer Science, Springer, 1995, pp. 307–323.

[155] Z.-D. Shen, F. Yan, W.-Z. Qiang, X.-P. Wu, and H.-G. Zhang, *Grid system integrated with trusted computing platform*, in Computer and Computational Sciences, 2006. IMSCCS '06. First International Multi-Symposiums on, vol. 1, june 2006, pp. 619 –625.

[156] R. Shirey, *Internet Security Glossary, Version 2*. RFC 4949 (Informational), August 2007.

[157] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The hadoop distributed file system*, in Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, Washington, DC, USA, 2010, IEEE Computer Society, pp. 1–10.

[158] A. Smith, *Proof of Theorems in Z*, tech. rep. No.: 92006, Defence Research Academy, Malvern, 1992.

[159] J. M. Spivey, *The Z notation: a reference manual*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[160] J. M. Spivey, *fUZZ Manual*, tech. rep., University of Oxford Computing Laboratory, 2008.

[161] S. Srirama, O. Batrashev, and E. Vainikko, *Scicloud: Scientific computing on the cloud*, in Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, Washington, DC, USA, 2010, IEEE Computer Society, pp. 579–580.

[162] Sufatrio and R. Yap, *Extending ban logic for reasoning with modern pki-based protocols*, in Network and Parallel Computing, 2008. NPC 2008. IFIP International Conference on, Oct. 2008, pp. 190–197.

[163] Trusted Computing Group (TCG), *TCG infrastructure working group: Architecture part ii - integrity management*. http://www.trustedcomputinggroup.org/files/resource_files/87651761-1D09-3519-AD6C5B3E41547285/IWG_ArchitecturePartII_v1.0.pdf.

[164] Trusted Computing Group (TCG), *TCG official website*. http://www.trustedcomputinggroup.org.

[165] Trusted Computing Group (TCG), *Trusted Platform Module (TPM) Specifiation*, 2008.

[166] D. Thain, T. Tannenbaum, and M. Livny, *Condor and the grid*, in Grid Computing: Making the Global Infrastructure a Reality, F. Berman, G. Fox, and T. Hey, eds., John Wiley & Sons Inc., December 2002.

[167] The MITRE Corporation, *Common Attack Pattern Enumeration and Classification (CAPEC) web site*. `http://capec.mitre.org`, July 2012.

[168] The MITRE Corporation, *Common Weakness Enumeration (CWE) web site*. `http://cwe.mitre.org`, July 2012.

[169] A. Ulrich, H. Hallal, A. Petrenko, and S. Boroday, *Verifying trustworthiness requirements in distributed systems with formal log-file analysis*, Hawaii International Conference on System Sciences, 9 (2003), p. 337b.

[170] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, A. Rozinat, E. Verbeek, and T. Weijters, *Prom: The process mining toolkit*, in BPM (Demos), A. K. A. de Medeiros and B. Weber, eds., vol. 489 of CEUR Workshop Proceedings, CEUR-WS.org, 2009.

[171] W. M. P. van der Aalst, T. Weijters, and L. Maruster, *Workflow mining: Discovering process models from event logs*, IEEE Trans. on Knowl. and Data Eng., 16 (2004), pp. 1128–1142.

[172] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst, *The prom framework: A new era in process mining tool support*, in Application and Theory of Petri nets 2005, vol. 3536, Springer, Berlin, Germany, 2005, pp. 444–454.

[173] C. Vishik, A. Rajan, C. Ramming, D. Grawrock, and J. Walker, *Defining trust evidence: research directions*, in Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research, CSIIRW '11, New York, NY, USA, 2011, ACM, pp. 66:1–66:1.

[174] webinos Consortium, *webinos web site*. `http://webinos.org`, March 2012.

[175] T. White, *Hadoop: The Definitive Guide*, O'Reilly Media, Inc., 1st ed., 2009.

[176] P. Y. H. Wong and J. Gibbons, *A process semantics for BPMN*, in Formal Methods and Software Engineering, S. Liu, T. Maibaum, and K. Araki, eds., vol. 5256 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 355–374. 10.1007/978-3-540-88194-0_22.

[177] J. C. P. Woodcock, *Structuring specifications in z*, Softw. Eng. J., 4 (1989), pp. 51–66.

[178] J. C. P. Woodcock and J. Davies, *Using Z: specification, refinement, and proof*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[179] D. Xiu and Z. Liu, *A formal definition for trust in distributed systems*, in Information Security, J. Zhou, J. Lopez, R. Deng, and F. Bao, eds., vol. 3650 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005, pp. 482–489. 10.1007/11556992_35.

[180] T. Yamagishi and M. Yamagishi, *Trust and commitment in the United States and Japan*, Motivation and Emotion, 18 (1994), pp. 129–166. 10.1007/BF02249397.

[181] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, *Discotect: a system for discovering architectures from running systems*, in Software

Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on, 2004, pp. 470–479.

[182] F. Yuan, S. Wright, K. Eder, and D. May, *Managing complexity through abstraction: A refinement-based approach to formalize instruction set architectures*, in 13th International Conference on Formal Engineering Methods (ICFEM), Lecture Notes in Computer Science 6991, October 2011, pp. 585–600.

[183] J. A. Zachman, *A framework for information systems architecture*, IBM Syst. J., 26 (1987), pp. 276–292.