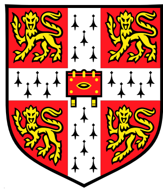


Trust Management for Widely Distributed Systems

Walt Teh-Ming Yao

Jesus College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

February 10th, 2003

Abstract

In recent years, we have witnessed the evolutionary development of a new breed of distributed systems. Systems of this type share a number of characteristics. They are highly decentralized, of Internet-grade scalability, and autonomous within their administrative domains. Most importantly, they are designed to operate collaboratively, regardless of whether they know each other or not. Among many applications, the prime examples of this type of distributed systems include peer-to-peer systems and web services.

Traditionally, authorization in distributed systems is mostly identity-based – a natural evolution from classic multiuser time-sharing (MTS) systems such as Unix. For the new breed of distributed systems, this approach nevertheless is insufficient in several areas: (1) the need to deal with unknown users, i.e. strangers; (2) the need to manage an exceedingly large number of users and/or sizable resources; (3) natural mapping of organizational policies into security policies; (4) managing collaboration of independently administered domains/organizations; (5) decentralizing security policies and policy enforcement.

In order to address the above mentioned areas, a capability-style, policy-driven, *trust management framework* named Fidelis has been devised. A trust management system is a unified framework for the specification of security policies, the representation of credentials, and the evaluation and enforcement of policy compliances. Fidelis is designed specifically for widely distributed applications. Based on the concept of *trust conveyance* and a generic abstraction for trusted information as trust statements, Fidelis provides a general platform for building secure, trust-oriented distributed applications. At the heart of Fidelis is a language for the specification of security policies, called the Fidelis Policy Language (FPL). With its uniform treatment of both trust statements and principals, recommendation-style policies may be naturally expressed, and arbitrarily complex chains of trust may be modelled.

Web services have been considered by the industry and researchers as the ubiquitous, next-generation middleware platform. The second half of the thesis describes the design and implementation of Fidelis for the emerging web service platform. The two primary intentions for this architecture are: first, to demonstrate the practical feasibility of Fidelis, and second, to investigate the use of a policy-driven trust management framework for Internet-scale open systems. A trust negotiation framework is devised as an enabling mechanism for unfamiliar principals to establish trust, and consequently for them to engage in trust-directed transactions. The framework integrates with Fidelis, and the negotiation behaviours can be directed by policies specified in FPL.

The applicability of Fidelis is far-reaching. Case studies are presented to examine three distinctive domains: providing role-based access control, trust management in the World Wide Web, and an electronic marketplace comprising unfamiliar and untrusted but collaborative organizations.

To my parents Kai-Lin and Mei-Lun

Preface

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

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

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

This dissertation does not exceed 60,000 words including tables and footnotes, but excluding bibliography and diagrams.

This dissertation is copyright ©2002 by Walt Teh-Ming Yao.
All trademarks used in this dissertation are hereby acknowledged.

Acknowledgements

This work would have not been possible without the continuous support, advice, and encouragement from my supervisor, Jean Bacon. Throughout my PhD life, she has always been tireless in giving me invaluable comments and advice. Asides from work, she has also been exceptionally understanding and sympathetic with other problems I have encountered during this time, especially the period when I was recovering from an eye operation. I express my greatest appreciation toward her guidance throughout the period of my study.

I am also grateful to Ken Moody, who has provided me with constant critical discussions that were both constructive and inspiring. Without his advice, I would not have tackled many of the problems that I encountered during my research.

Many thanks are also due to the fellow researchers in the OPERA research group at the Computer Laboratory. They have all been fun to work with, to play with, and to learn from. Working with them has given me numerous intellectual, interesting and enjoyable discussions, both academically and leisurely. I shall pay extra appreciation to those who proof-read my thesis and corrected many language problems – David Ingram, Brian Shand, Nathan Dimmock and especially David Eyers, for his extraordinarily high-quality comments.

I would also like to thank the UK Engineering and Physical Research Council (EPSRC) for supporting this work, under the grant *OASIS Access Control: Implementation and Evaluation*.

I owe a special debt to my parents, Yao Kai-Lin and Fang Mei-Lun, not only for financially supporting my ten-year study in the UK, from school through to doctoral studies; but also for their endless care, understanding, support, and advice on my personal life.

List of Publications

- John Hine, Walt Yao, Jean Bacon, and Ken Moody. An architecture for distributed OASIS services. In *Middleware 2000 (Palisades, NY, April 4–8)*, volume 1795 of *Lecture Notes in Computer Science*, pages 104–120, Heidelberg, Germany, April 2000. Springer-Verlag.
- Jean Bacon, Alexis Hombrecher, Chaoying Ma, Ken Moody, and Walt Yao. Event storage and federation using ODMG. In *Proc. 9th International Workshop on Persistent Object Systems (POS9, Lillehammer, Norway Sept. 6–8)*, volume 2135 of *Lecture Notes in Computer Science*, pages 265–281, Heidelberg, Germany, September 2000.
- Walt Yao, Ken Moody, and Jean Bacon. A model of oasis role-based access control and its support for active security. In *Sixth ACM Symposium on Access Control Models and Technologies (SACMAT 2001, Chantilly, VA, May 3–4)*, pages 171–181, New York, NY, May 2001. ACM Press.
- Jean Bacon, Ken Moody, and Walt Yao. Access control and trust in the use of widely distributed services. In *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 300–315, Heidelberg, Germany, November 2001. Springer-Verlag.
- Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5(4), November 2002. To appear.

Contents

1 Introduction **1**
1.1 Distributed authorization and trust management 2
1.2 New challenges 3
1.3 Research issues 3
1.4 Thesis contribution 4
1.5 Security engineering 5
1.6 Dissertation outline 6

2 Related Work **9**
2.1 Access control models 9
2.1.1 Mandatory access control (MAC) 10
2.1.2 Clark and Wilson model 11
2.1.3 Chinese Wall policy 12
2.1.4 Discretionary access control (DAC) 13
2.1.5 Role-based access control (RBAC) 14
2.2 Distributed access control 16
2.2.1 Access control lists in distributed systems 16
2.2.2 Capability-based access control 17
2.2.3 Credential-based access control 18
2.2.4 Categories of credential-based access control 19
2.3 Identity-oriented access control 20
2.3.1 X.509 Public Key Infrastructure 21
2.3.2 Pretty Good Privacy (PGP) 22
2.3.3 Attribute certificates 23
2.4 Key-oriented access control 24
2.4.1 Simple Public Key Infrastructure (SPKI) 25
2.4.2 PolicyMaker and KeyNote 27
2.4.3 Other trust management systems 29
2.5 Summary 31

3 Fidelis Trust Management Infrastructure **33**
3.1 Overview of the Fidelis Trust Management Infrastructure 33
3.2 Trust model 34
3.2.1 Trust as a security concept 35
3.2.2 Trust as a sociological concept 36
3.2.3 The basis of trust 38
3.3 Conveying trust 39
3.3.1 Basic concept 39
3.3.2 Validity 40
3.3.3 Discussion 41
3.4 Identity 41
3.4.1 Discussion 42

3.5 The Fidelis Policy Language . . . 43
3.5.1 Principals . . . 43
3.5.2 Actions . . . 44
3.5.3 Trust specification . . . 45
3.5.4 Validity conditions . . . 46
3.5.5 Trust relationships . . . 47
3.5.6 Action policies . . . 53
3.5.7 Conditional and assignment expression . . . 54
3.5.8 Evaluation semantics . . . 55
3.5.9 Discussion . . . 57
3.6 Summary . . . 58

4 Fidelis and Web Services **61**
4.1 Introduction . . . 61
4.1.1 Background . . . 61
4.1.2 Design issues . . . 63
4.2 Service architecture . . . 64
4.2.1 Locating principals . . . 64
4.2.2 Conveyance interface . . . 65
4.2.3 The trust inference interface . . . 66
4.2.4 The credential management interface . . . 68
4.2.5 The policy interrogation interface . . . 69
4.2.6 The trust agent interface . . . 71
4.2.7 Identifying requesters . . . 73
4.3 Fidelis Policy Interchange . . . 74
4.3.1 Overview . . . 74
4.3.2 The top-level container . . . 75
4.3.3 Schema definitions . . . 75
4.3.4 Principal declarations . . . 76
4.3.5 Policy specification . . . 77
4.3.6 Linking with other policy documents . . . 79
4.4 Credential representation . . . 79
4.4.1 Basic structure . . . 79
4.4.2 Truster and subject . . . 80
4.4.3 Validity condition . . . 81
4.4.4 Signature . . . 81
4.5 Summary . . . 82

5 Inference and Trust Negotiation **83**
5.1 Policy inference . . . 83
5.1.1 Inference algorithm . . . 83
5.1.2 Managing distrust repositories . . . 88
5.1.3 Tracking validity . . . 89
5.1.4 Runtime analysis . . . 90
5.2 Trust negotiation . . . 92
5.2.1 Trust negotiation overview . . . 92
5.2.2 Trust negotiation protocol . . . 93
5.2.3 Meta policies . . . 94
5.2.4 Related work . . . 97
5.3 Summary . . . 98

6 Applications **99**
6.1 Role-based access control . . . 99
6.1.1 OASIS role-based access control . . . 99
6.1.2 RBAC96 and the NIST unified model . . . 102
6.1.3 Discussion . . . 106
6.2 Case study: Trust management in the World Wide Web . . . 107
6.2.1 Architectural overview . . . 107
6.2.2 Request handling in Apache . . . 108
6.2.3 Integrating Fidelis . . . 109
6.2.4 Discussion . . . 113
6.3 Case study: an electronic marketplace . . . 113
6.3.1 Background . . . 113
6.3.2 Environment . . . 114
6.3.3 Membership management . . . 115
6.3.4 Product catalogue management . . . 115
6.3.5 Reputation management . . . 116
6.3.6 Transaction processing: purchases . . . 117
6.3.7 Discussion . . . 118
6.4 Summary . . . 118

7 Discussion **121**
7.1 Policy framework . . . 121
7.2 Managing scalability . . . 122
7.3 Decentralized collaboration . . . 123
7.4 Privacy . . . 124
7.5 Decentralization approaches . . . 125
7.6 Summary . . . 126

8 Conclusions and Future Work **127**
8.1 Summary of contributions . . . 127
8.2 Future work . . . 128
8.3 Conclusion . . . 128

List of Figures

1.1 Framework for security engineering . . . 6

2.1 An access control matrix . . . 13
2.2 A basic RBAC model . . . 14
2.3 An example of role hierarchy . . . 15
2.4 A certification path . . . 21
2.5 Examples of trust model . . . 22
2.6 PMI delegation model (simplified from [6]) . . . 24
2.7 Sample KeyNote assertion . . . 29

3.1 Fidelis overview . . . 34
3.2 Conveying trust . . . 40
3.3 Transitive trust and delegating trust . . . 53
3.4 Examples of regular expression patterns . . . 55

4.1 A sample SOAP message (message content from [7]) . . . 62
4.2 Trust inference - action decision . . . 68
4.3 Automated credential collection . . . 69
4.4 Policy discovery . . . 71
4.5 Assisted request initiation through a trust agent. . . . 72
4.6 Trust negotiation between principals. . . . 73
4.7 `@method` URI identifiers for online validity schemes. . . . 81

5.1 Passive replication scheme . . . 89
5.2 Validity dependency tree . . . 90
5.3 A trust negotiation session . . . 92
5.4 State diagram for the negotiation protocol . . . 94
5.5 Vocabulary for the meta-policy profile . . . 95

6.1 An example role hierarchy (adopted from [8]). . . . 103
6.2 Role memberships for users in the examples. . . . 104
6.3 Proxy mechanism . . . 107
6.4 Request handling cycle in the Apache server (version 1.x) . . . 108
6.5 Architecture of `mod_fidelis` . . . 109
6.6 Commonly-used CGI variables. . . . 111
6.7 Incorporating product information . . . 115
6.8 Supporting purchase decision. . . . 117
6.9 Delegated purchase . . . 118

1 Introduction

With the growing popularity of the Internet, open, large-scale distributed applications are becoming increasingly prevalent. While past research on authorization for distributed systems has addressed many issues in traditional networking/distributed systems, today's open and highly decentralized applications have raised many new questions in the unexplored territories in the distributed systems security research.

Numerous attempts have been made in the past to apply traditional, well-studied authorization schemes to cope with the needs of distributed systems. Most of these attempted to extend identity- or capability-based systems, combining them with cryptographic authentication protocols. However, such efforts often only address a partial set of the outstanding issues. In recognition of the failings of the conventional approaches, Blaze et al. [9] proposed the *trust management approach* to decentralized authorization management. The basis of their trust management approach centres around the notion of *delegation certificates* – capability-like credentials. Every delegation certificate delegates some authorization from its issuer to its subject; chains of certificates issued by different issuers may be formed, thus enabling authorization to be granted in a decentralized manner.

The key concept advocated by the trust management approach is the holistic treatment of distributed authorization management, with a unified framework for the management of security policies, security credentials and trust relationships. While this represents a major advance for distributed systems security, departing from traditional approaches, there are still many issues yet to be resolved. Unlike traditional networked services, today's distributed services must face the new challenges posed by an open network. Firstly, the scale of the system with the sheer number of potential users and sizable resources makes obsolete the possibility of centralized security management. Decentralized administration is no longer just an option but indeed a necessity to address scalability problems. Secondly, collaboration among strangers in an open system becomes unavoidable: competing organizations may be required to cooperate; businesses with conflicting goals may need to collaborate; a person may need to shop at an online store at which she has never been before. Thirdly, there is typically a lack of a single authority that is unanimously trusted and agreed upon by all parties. Each party in the network is assumed to have full autonomy to specify, enforce, and monitor its own security policies and mechanisms.

This thesis presents my research on the topic of distributed authorization management, especially for the aforementioned new styles of distributed applications. Based on the trust management approach, we are addressing issues previously unresolved by the current state-of-the-art with a new trust management system, called *Fidelis*. Fidelis is designed and implemented as part of this research, and features a fully decentralized and policy-driven framework.

This chapter describes the motivation and outlines the contribution of this work. It begins by briefly reviewing the state-of-the-art in authorization management for distributed systems in Section 1.1. Section 1.2 examines new challenges posed by the new types of distributed applications we have mentioned. This is followed by a summary of pending research issues to date in Section 1.3. Section 1.4 outlines the contribution of this research. Section 1.5 describes a layered approach to security engineering. This layering is reflected in the structure of this thesis, which is described in Section 1.6.

1.1 Distributed authorization and trust management

Traditional approaches to distributed authorization are generally either identity-based or capability-based. This is to be expected, as they have had a natural evolution from the security research in operating systems, later being extended to cater for networked and distributed applications. The identity-based approach focuses on authentication. The idea is that a requester to a distributed service needs to be securely authenticated before an access decision can be made using conventional schemes, such as access control lists. Identity-based authorization stimulated much research on cryptographic protocols [10, 11, 12] that allow communicating parties to identify each other and often also establish a shared secret for securing communication sessions.

Capability-based systems such as described by Gong [13], Bull et al. [14] and Hayton [15] take a different approach. Instead of relying on requester identity, these systems rely on capabilities contained in *credentials* to grant or deny access. Management of credentials is therefore the focus in the capability-based approach. A variety of techniques have been developed for this: some employ cryptography to prevent theft and forgery, while others devise architectures to ensure controlled transfer of credentials. In comparison with the identity-based approach, capability-style authorization is more suited for distributed systems, as it encourages distributed security management and is hence inherently more scalable.

Authorization management attempts to address a whole spectrum of issues, ranging from the high-level organizational policies, through the specification of security policies, to low-level security mechanisms. Both approaches described above typically only address a subset of these issues and, as a result, do not always satisfy precise application needs. Trust management is an alternative approach that aims at delivering a unified framework for managing security policies, credentials and their trust relationships. Based on concepts pioneered by capability systems, a trust management system attempts to answer authorization questions in the form of "*is a request r compliant with the local policies P given the set of credentials C?*". A crucial element of trust management is the consideration of security policies, which was merely supported at a lower level with traditional approaches. A trust management system can be broken down into three basic components [9, 16]:

- A language for expressing security policies. This includes the means of describing actions, identifying principals, and specifying trust relationships.
- A language for specifying security credentials, which may be transferred between entities in the system to express delegation of authority.
- A *compliance checker*, which computes whether a request should be granted given the local policies and a set of credentials. This is also commonly known as the *trust management engine.*

One of the key features of existing trust management systems is to decentralize policy management based on *delegation of authority.* For example, a resource provider may delegate the rights of accessing its resource to some principal through a digital credential. That principal may in turn delegate this right to another principal, and this process may proceed indefinitely. The ultimate principal may present the set of credentials at the resource provider, where its compliance checker will then attempt to find a *chain of delegation* from the set to make authorization decisions.

Delegation of authority is not unique to trust management. It also forms the basis of *key-oriented access control*, whose representative systems include the Simple Distributed Security Infrastructure (SDSI) [17] and the Simple Public Key Infrastructure (SPKI) [18]. A core concept of these new access control systems is the first-class treatment of public keys as principal identifiers, and naturally relies on the use of public key cryptography to provide principal authentication. While the focus of such systems is not on the design of a unified security framework, they may be considered as a form of trust management system, because of their well-defined compliance computation [16].

Although the current, state-of-the-art, distributed authorization is a major improvement over the traditional approaches, today's modern distributed applications generate new requirements that need to be addressed. In the next section, we will discuss properties of these new applications and their relation to authorization management.

1.2 New challenges

The advances in communications, networking and middleware research have brought distributed systems to new prominence. With the global reach of the Internet, widely distributed applications are increasingly commonplace. Some of their major characteristics may be observed:

- *Internet-scale*

  New applications are required to potentially scale up to the scope of the Internet, implying the need to manage vast resources and numbers of distributed users from anywhere in the world. The authorization framework, as a critical part of any trust reliant application, evidently must be as scalable as the application itself.

- *Cross-boundary*

  Because of the scale, new distributed applications often span several network, administrative or organizational boundaries. For example, an enterprise resource planning system (ERP) for a multi-national organization may need to integrate several geographically dispersed sites under one application. The authorization framework must support cross-boundary management and administration.

- *Autonomous*

  Closely related to the previous points, it is generally difficult, costly, and/or cumbersome to impose a central authority when applications span several boundaries. Each administrative domain should hence be assumed to have full autonomy of specification, management and enforcement of its security policies. The authorization framework must have support for inter-linked and inter-operating autonomous domains.

- *Open*

  Modern distributed applications tend to be highly open in nature. For example, a peer-to-peer file-sharing application allows virtually any Internet users to interact with each other; a web-based online store is open to everyone. This implies that applications are often required to deal with previously unknown or unfamiliar principals. The authorization framework must be designed to handle strangers in compliance with the application security policies.

- *Complex authorization policy*

  Traditional authorization mechanisms typically only consider simple attributes such as the user-name or clearance level. In today's applications, we often observe the need to express complex (and relatively high-level) policies. For example, a user of a peer-to-peer file sharing program may only wish to share her files with people who are either: her friends, have uploaded 10MB of files in exchange, or anyone if it is between 1am to 7am. The authorization framework should be sufficiently flexible and expressive to support complex policies.

- *Evolution*

  Because of the scale, changes to an application often cannot be made atomically as a "big-bang". As the application evolves, the security policies will need to evolve accordingly. Ideally, the authorization framework should support incremental deployment, and to a certain extent, must co-exist with legacy security mechanisms.

Having presented these new challenges for authorization frameworks, we are in a position to highlight the research issues raised in addressing them.

1.3 Research issues

The main research issues raised by modern distributed applications which yet remain unresolved by the current state-of-the-art of distributed authorization include:

- *Policy framework*

  Traditional identity-based or capability-based authorization focus on the mechanisms enforcing security. While the modern trust management approach is more inclined to the policy support than the enforcement mechanism, current solutions lack comprehensive frameworks for policy specification. For example, PolicyMaker [9], the most well-known trust management system, and its successor KeyNote [19, 20] feature *programmable credentials*, where policies are expressed as programs to be executed by a trust management engine. While this achieves unparalleled expressiveness, it makes policy specification, management and maintenance difficult tasks. An ideal approach would be based on a policy framework backed by a clearly defined model and processing semantics.

- *Managing scalability*

  As previously discussed, today's distributed applications often need to face heightened scalability requirements, meeting the demand of the Internet. While the trust management approach, due to its capability-like nature, has some degree of support for decentralization built-in, many improvements still need to be made in order to meet the rigorous scalability requirements. For example, current trust management systems assign privileges directly to identities with credentials. If the security policies evolve, old credentials need to be revoked while new ones are issued. This task becomes prohibitively expensive as the number of credentials becomes exceedingly large. A possible solution is to integrate elements of role-based access control (RBAC) into the trust management framework.

- *Decentralized collaboration among unfamiliar parties*

  The openness of new distributed applications consequently results in communication and collaboration with strangers. There has been a lack of attention in this area by the current trust management systems. Most current systems, while decentralized, assume the issuer and the acceptor of a credential share common vocabularies. In a truly open environment, dynamic trust negotiation is often required for two previously unknown parties to gradually gain trust and subsequently be engaged in a collaboration or transaction.

- *Privacy*

  Many of the current trust management systems adopt the idea of first-class treatment of public keys, i.e. public keys as principal identifiers without compulsory linkage to private data. While this offers a potential platform for implementing pseudonymous communications, none of the current trust management systems are designed with protection of privacy in mind. Ideally, a trust management framework should have provisions for anonymous or pseudonymous communication, while it is an application issue whether such features are utilized.

- *New approaches to decentralization*

  Existing trust management systems are based strongly on the concept of *delegation of authority*. While delegation of authority is important and indeed should be supported, other types of decentralization may exist. It remains an active research topic to examine other possible decentralization techniques, in particular the structuring of authority.

The research issues discussed in this section effectively set out the goals for this work. This thesis is intended to address most of the above mentioned issues in an attempt to devise a viable trust management system for Internet-scale distributed applications.

1.4 Thesis contribution

The main contribution of the thesis is the design and implementation of a novel, fully policy-driven trust management framework – *Fidelis*. Fidelis is designed to address many of the pending research issues described in the previous section. The list of contributions is described below, together with the chapters where the relevant work is found.

- Proposing a generic model capturing the essence of a trust management system. The model is called the *trust conveyance model*, and is described at an abstract level, with the intention to serve as a general foundation for future trust management systems, including, but not limited to, Fidelis. (Chapter 3)
- Designing a policy framework realizing the trust conveyance model and featuring *attribute-based* trust authorization management. Attributes with their meta-data are shown to be able to express arbitrary statements and actions. A policy language called the *Fidelis Policy Language* (FPL) is presented for the specification of trust statements, actions and their relationships. An important part of this work is the specification of the semantics for the FPL trust computation. (Chapter 3)
- Designing and implementing Fidelis for the web-service platform. This involves several aspects, ranging from the architecture, the interfaces, the protocol, and the message/document format. The design and implementation are focused on two key principles: *interoperability* and *practical applicability*. The aim is to produce a trust management platform on which real applications may be built. (Chapter 4)
- Designing and implementing an algorithm for the computation of trust compliance, strictly conforming to the evaluation semantics defined as part of the policy framework. The algorithm mainly serves as a proof-of-concept for the viability of the policy framework. (Chapter 5)
- Designing and implementing a trust negotiation framework. The trust negotiation framework is equipped with a flexible policy control of the negotiation process, by applying Fidelis. This negotiation framework is designed specifically for two purposes: as a demonstration of the applicability of Fidelis, and as a platform to enable communication between complete strangers – a scenario commonly-encountered in today's distributed applications. (Chapter 5)
- Experimenting with and studying the use of Fidelis in a number of application domains. This provides an insight into the effectiveness of the trust management approach under the demanding requirements of today's applications. Through this study, some tools and technologies have been developed which may be deployed in a wider context. For example, a module has been implemented to allow Fidelis to be integrated seamlessly with the Apache web server. (Chapter 6)

1.5 Security engineering

As discussed in previous sections, security problems for future distributed applications present complex research challenges. These problems cannot be tackled in a single step due to their complexity and inter-relationships. A good, well-known engineering practice is to divide a large problem into smaller pieces, solve each piece separately, and correlate individual solutions to produce a consistent solution. For security engineering, the same technique applies. A layered approach to security engineering has been proposed in [21]. It separates security issues into four layers, shown in Figure 1.1.

Policy states the high-level organizational goals and requirements. It is driven by the anticipated threats and goals, and considers the principles of risk management. It is usually concisely and formally written in natural language.

Model decomposes policies into abstract terms that can be analyzed and mapped into implementable entities. This often takes the form of formal, rigorous mathematical descriptions, but sometimes precise use of natural language is sufficient.

Architecture describes high-level security designs in terms of the major components of a system and their inter-relationships. In an operating system, this includes the memory protection module, the file system, etc; whereas in a distributed system, this would instead include servers, databases, middleware, etc.

Mechanism is a set of means to implement the security design. For a multi-level security (MLS) system, these may be security labels and protected objects. For distributed systems, these may include network protocols, credentials, or tickets.

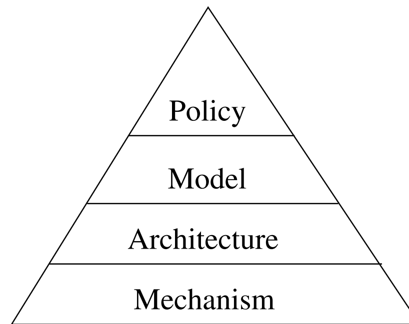


Figure 1.1: Framework for security engineering

The top two layers of the pyramid, namely policy and model, are concerned with formulating *what* the security requirements, relevant issues and trade-offs are, while the bottom two layers focus on *how* these requirements can be met. The inter-relationships between the layers may often be complex and thus inappropriate for a top-down design process such as the waterfall method (used in software engineering). Instead, an iterative refinement approach is more suitable as, for example, the implications of the chosen mechanisms must be taken into account in the layers above, and the effects of a change of an objective on other layers must be fully analyzed and incorporated.

Security is a qualitative and holistic property of the system which must be considered as a whole. It is therefore important to take into account all four layers to produce a consistent security framework. The work described in this thesis follows the layered approach. Each layer is addressed separately but with cross-layer inter-relations discussed. This is reflected in the structuring of this thesis itself, as summarized in the next section.

1.6 Dissertation outline

This thesis is organised as follows:

Chapter 2 reviews major work in the area of authorization management, with the focus on distributed systems. It begins with an overview of the general access control problem, followed by descriptions of various distributed authorization schemes in two categories: identity-oriented and key-oriented. The chapter ends with a comprehensive review of the state-of-the-art in trust management systems.

Chapter 3 introduces the Fidelis trust management infrastructure. Prior to the description of Fidelis, the notion of *trust* in the literature is discussed. The intention here is to form a solid basis for Fidelis. Fidelis is described in two parts in this chapter: the conceptual model and the policy framework. The model describes the fundamental model – the *trust conveyance model*. The policy framework concentrates on the description of the Fidelis Policy Language.

Chapter 4 describes an implementaion of the Fidelis trust management framework for the web service environment. This covers the architectural design applying the recent web service technologies. It also describes two additional pieces of technology which are designed to facilitate interoperation between any pair of locally autonomous principals in the global web-service network: the Fidelis Policy Interchange and the Fidelis Interoperable Credential format.

Chapter 5 describes an algorithm that implements the trust compliance semantics defined in Chapter 3. This algorithm is designed to demonstrate that implementations of the semantics exist. It does not, however, exploit possible optimizations. The second part of this chapter describes a trust negotiation model that is designed to enable complete strangers to incrementally learn about each other and eventually collaborate.

Chapter 6 provides in-depth descriptions of several applications built to employ Fidelis as their authorization mechanism. These applications aim to demonstrate various specific features of Fidelis in practice. Among them, a case study of electronic commerce is included, which attempts to gain practical insight into this application domain, and to evaluate this work.

Chapter 7 provides a critical evaluation of this work against the goals set out in Section 1.3. The evaluation is qualitative in the form of discussion and is largely based on experience gained while designing and implementing the test-case applications described in Chapter 6.

Chapter 8 concludes this thesis, with a summary of the main contributions and a brief discussion of potential future research and extensions.

2 Related Work

The concept of trust management is closely related to that of access control. The trust management approach is essentially distributed access control with extensions relating to *trust*, e.g. the notion of trust expression, trust propagation and trust-directed security policies. This chapter reviews major work in the area of access control, with a focus on distributed access control.

The use of the term *access control* in this thesis includes both the notion of *authentication* and *authorization*. Authentication is concerned with securely identifying subjects, while authorization addresses the granting of access rights once a subject has been authenticated.

This section starts by reviewing access control models. The concept of access control models historically originates from the study of *security policies*, which can be briefly described as a set of requirements, properties and mechanisms to protect resources in a system. Section 2.1 introduces some influential models, including mandatory access control policies, the Clark and Wilson model, the Chinese Wall policy, and the role-based access control model.

It then describes the concept of access control in distributed systems. The review starts from early work on distributed capability systems, and continues to the credential-based approach. It then introduces two categories of credential-based access control: identity-oriented and key-oriented. Section 2.3 describes the major work based on the concept of identity-oriented credentials, notably the ITU/ISO X.509 Public Key Infrastructure. Section 2.4 describes work based on the new key-oriented approaches for access control, in particular a number of *trust management systems* are described.

2.1 Access control models

One branch of the early work on access control models came from the study of *security policies* in the military sector in the 70s, and another came from the research on operating systems security. In this section, we shall concentrate on the former, while the latter will be described in the context of distributed systems in Section 2.2.

The primary concern of military systems is confidentiality of data, where prevention of information leakage is the most important goal. In response to this need, Bell and LaPadula [22] introduced a security model based on the military-style clearance scheme that restricts flows of classified information. Their work lead to the development of numerous multilevel security (MLS) systems, and is arguably one of the most influential models in the history of computer security.

While confidentiality is by far the most crucial requirement in military systems, integrity of data is conceived to be equally or even more important in the commercial sector. Significant attempts to model integrity requirements include Biba [23] and Clark and Wilson [24]. The latter, in particular, represents an influential shift of focus from military-oriented security policies to commercial ones in the 80s. It formalized well-established business practices of double entry bookkeeping and separation of duty, and proposed an abstract model and mechanisms to enforce such rules. Business relationships often cause conflicts of interest between different parties, for example when a consultant is providing services to competing businesses. To model conflicts of interest intrinsic in business relationships,

Brewer and Nash [25] introduced the Chinese Wall security policy, which prevents the breach of confidentiality by *insider knowledge* through consideration of access histories.

The 90s saw a growing interest in *role-based access control* (RBAC). First formalized by Ferraiolo and Kuhn [26] in 1992, RBAC is primarily based on the observations that previous access control models for the military and commercial sectors often do not naturally reflect higher-level organizational policies. The basic idea is that a role reflects an organizational job function and the concept of roles is seen as a natural unit to model policies, acting as a bridge between security mechanisms and policies.

In general, an access control model is a set of formalized, concise security goals and properties, plus abstract mechanisms for enforcing them. This section reviews the above mentioned models and policies in more detail.

2.1.1 Mandatory access control (MAC)

Multilevel security (MLS) policy and mechanism were developed in the military as a means to manage classified information. Each document is labelled with a degree of sensitivity, known as a *classification* e.g. "unclassified", "confidential", "secret" and "top-secret". All military personnel are assigned a *clearance* level on the same labelled scale as the classification. This assignment may depend on a variety of factors, including ranks, units, etc. The access control policy states that an officer must have a clearance at least as high as the classification of the document he/she attempts to read. The safety of this system comes from the strict one-way information flow, i.e. information may only flow upwards in the sensitivity scale but never downwards, unless it is explicitly declassified by an authorized person. The term *Mandatory Access Control* (MAC) is defined by the United States Department of Defense Trusted Computer System Evaluation Criteria, the "Orange Book", as "a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (e.g., clearance) of subjects to access information of such sensitivity".

The seminal attempt to formalize the multilevel security policy was due to Bell and LaPadula, and their formalism is often referred to as the Bell-LaPadula or BLP model [22]. The primitive elements in BLP are *subjects*, *objects*, *access rights* and *security levels*. The set of access rights contains mainly two operations, *read* and *write* access. A security level is defined as a tuple consisting of a *classification* and a set of *categories*. The set of classifications contains names ordered by a $>$ relation, e.g. *top-secret*, *secret*, etc. The set of categories contains names describing compartments, such as *NATO* and *nuclear*. Each object is associated with a security level, denoting its degree of sensitivity. Each subject is associated with a maximum security level and a current security level, which can be changed dynamically if necessary.

A binary, partial-order relation *dominates* is then defined between a pair of security levels, a and b in such way that,

$$\begin{aligned}\forall a, b \in levels, a \ dominates\ b \quad \Longleftrightarrow \quad & classification(a) \geq classification(b) \wedge \\ & categories(a) \supseteq categories(b)\end{aligned}$$

For instance, (*top-secret,* $\{NATO,$ *nuclear*$\}$) dominates (*secret,* $\{NATO\}$) because *top-secret* is higher than *secret* and $\{NATO\}$ is contained in $\{NATO,$ *nuclear*$\}$. However, (*secret,* $\{NATO\}$) does not relate to (*confidential,* $\{$*nuclear*$\}$). Two properties are then defined to express the security policy.

The *simple security property*, also known as "no read up", states that no subject may read objects at a higher level than his/her current level. Stated formally, a *read* access to an object is granted if and only if,

$$\forall s \in subjects, o \in objects : level(s)\ dominates\ level(o)$$

The **-property*, often called "no write down", states that no subjects may write to objects at a lower level than his/her current level. This is expressed formally that a *write* access to an object is granted if and only if,

$$\forall s \in subjects, o \in objects : level(o)\ dominates\ level(s)$$

The *-property was devised to address concerns of information leakage by malicious programs. For example, a Trojan horse that writes information to unclassified objects may be planted into a system by an unprivileged user. A privileged user may unknowingly execute it while reading classified information, which causes the information to be written to an unclassified object, effectively declassified. Lampson [27] introduced the *confinement problem*, which notes possible channels for information leakage, including storage, legitimate channels and *covert channels*. The *-property directly addresses the first type of channels by explicitly disallowing write access to objects with a lower security level than the subject.

Although the Bell-LaPadula model was designed to protect confidentiality of data, Biba [23] observed that a similar formulation could be applied to protect integrity. The Biba model is effectively the inverse of the BLP, i.e. high-integrity data should never be contaminated by low-integrity data. The information is restricted to flow from high-integrity to low-integrity. In particular, the model requires downgrading of a program if it reads lower level data to prevent possible contamination of data.

A final remark on the term *mandatory access control*. While the term historically refers to BLP-style, multilevel security policies, the intention behind the term is that the enforcement of the policy is independent of users' discretion or actions. Other access control policies such as the Clark-Wilson model to be described in the next section also exhibit mandatory behaviours. However, in order to be consistent with the terminology in the literature, this thesis uses the term mandatory access control to refer to multilevel security models.

2.1.2 Clark and Wilson model

Historically, research in access control policies has focused on guarding against unauthorized disclosure of information. This trend was driven by the needs of military environments, where confidentiality is the top priority. However, as noted by Clark and Wilson [24], in commercial systems, one of the primary objectives is the prevention of fraud and error. Fraud is typically achieved by unauthorized modification of information, while error typically causes inconsistency of information. Both these concerns can be addressed by enforcing integrity policies. It is therefore argued that integrity of information in such systems is more important than its confidentiality.

They presented a model, often referred to as the Clark-Wilson model, that formalizes two basic principles for achieving information integrity: *well-formed transactions* and *separation of duty*. These are derived from well-established mechanisms practised in business for centuries. The concept of well-formed transactions is that manipulation of data by a principal must be constrained in such a way that its integrity is preserved. A very common and effective mechanism employed in accounting is *double entry bookkeeping*. The idea is to record every single transaction twice, once in a book for credit and once in a book for debit. A later balance check would reveal discrepancies if any entry were not recorded correctly. The intention of well-formed transactions is to ensure the *internal* consistency and accuracy of the data.

The principle of separation of duty attempts to ensure *external consistency* where the data in the system reflect the real-world entities they represent, e.g. when a payment is recorded on the account as the fulfillment of a purchase, then there was indeed such a purchase, not a fraud. The correspondence to external entities is often abstract and hard to verify directly. The idea of separation of duty is to indirectly verify the correspondence to real-world entities by dividing a task among several principals. Provided these principals do not conspire, this mechanism should prevent both fraud and error.

The Clark-Wilson model partitions data into two sets: *constrained data items* (CDI), whose integrity must be ensured, and *unconstrained data items* (UDI), which are not under the control of the integrity policy, e.g. data input by a user from the keyboard. Two classes of procedures on these data items are defined to enforce the integrity policy: an *integrity verification procedure* (IVP) verifies

the integrity of all data items in the system, and a *transformation procedure* (TP) is a well-formed transaction that processes and changes a set of data items from one valid state to another.

The integrity policy can then be expressed in formalized rules, grouped into two types: *certification* and *enforcement*. Certification is an application-specific process that monitors the operations of a system with respect to a specific integrity policy. Enforcement rules are application-independent security functions that are automatically executed by the system. The rules in the Clark and Wilson model as formulated in [28] are:

Certification

- **C1 (IVP Certification)** For any CDI, there must exist some IVP on the system that validates its integrity.
- **C2 (Validity)** All TPs must be certified to maintain the validity of CDIs they processed.
- **C3 (Separation of Duty)** All possible operations on CDIs by potential users must be certified to implement the principles of separation of duties and least privilege.
- **C4 (Journal Certification)** All TPs must be certified to ensure sufficient logging for their operations.
- **C5** Special TPs that take UDIs must be certified to result in valid CDIs.

Enforcement

- **E1 (Enforcement of Validity)** Manipulation on a CDI must only be performed through a TP.
- **E2 (Enforcement of Separation of Duty)** Every user can only operate on a specific set of CDIs through a set of authorized TPs.
- **E3 (User Authentication)** Every user attempting to execute a TP must be properly authenticated by the system.
- **E4 (Initiation)** Only the administrator can specify authorizations to TPs and CDIs.

One of the main contributions of the Clark-Wilson model is that it offers a distinctive view of, and a set of mechanisms for, access control problems in commercial environments. Their work laid the groundwork for research in commercial security, such as the Chinese Wall policy described in the next section.

2.1.3 Chinese Wall policy

Brewer and Nash [25] introduced the Chinese Wall security policy to model the confidentiality constraints in the commercial sector to avoid *conflict of interest*. A classic example is a financial institution providing consultation services to business firms. Suppose the financial institution has clients from a variety of industries and there are several companies in each type of industry. If a market analyst working in the institution consults for one company, he/she cannot be permitted to consult for another company in the same industry, because the *insider knowledge* the analyst gains from one company may encourage unfair dealing for or against the other company.

In the Chinese Wall policy, protected objects of a company are grouped into a *company dataset*, and datasets from competing companies are grouped into a *conflict of interest class*. For an object o, $y(o)$ gives the name of the company where it belongs and $x(o)$ gives its conflict of interest class. Central to the Chinese Wall policy is the notion of *access history*, or *state*. This is kept in a two-dimensional matrix of Boolean values, N, with a column for each object and a row for each subject. An element $N_{s,o}$ is **true** if and only if subject s has previously accessed object o.

Modelling after the Bell-LaPadula (BLP) model, the Chinese Wall policy is formalized based on a simple security property and a *-property. The simple security property says that access to information by a subject is confined to one company of any given conflict of interest class. Specifically, access to object o is granted to subject s if one of the two following conditions is satisfied:

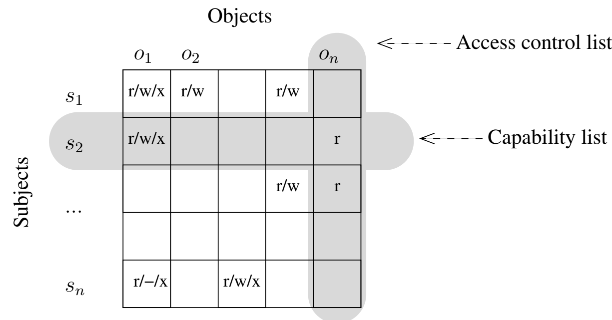


Figure 2.1: An access control matrix

- s has never dealt with any company of the conflict of interest class $x(o)$ in the past, i.e. for each object p such that $N_{s,p} = \textbf{true}$, $x(p) \neq x(o)$.
- s has dealt with the company $y(o)$ previously, i.e. for each object p such that $N_{s,p} = \textbf{true}$, $y(p) = y(o)$.

However, the simple security property alone is not sufficient to prevent information leakage. Suppose two subjects, Alice and Bob, are consulting for oil companies Shell and BP respectively and both at the same time are consulting for the HSBC bank. The simple security property does not stop Alice from writing confidential information about Shell to HSBC for Bob to read, thus indirectly violating the Chinese Wall policy.

The *-property addresses this type of violation. It states that write access of object o by subject s is only allowed if the simple security property is satisfied, and there does not exist any *unsanitized* object p previously read by s such that $y(p) \neq y(o)$. *Sanitization* is a transformation on an object that de-identifies its source so that disclosure of the sanitized object will not cause conflict of interest. The *-property ensures the confinement of the flow of unsanitized information to its own company dataset and allows sanitized information to flow freely within the system.

The Chinese Wall policy recognizes the importance of access history in protecting security and has made a seminal contribution to subsequent research on *history-based access control* and dynamic separation of duty in general [29, 30, 31, 32].

2.1.4 Discretionary access control (DAC)

The basic idea behind *discretionary access control* is that the owner of an object should be trusted to manage its security. More specifically, owners are granted full access rights to objects under their control, and are allowed to decide whether access rights to their objects should be passed to other subjects or groups of subjects at their own discretion; hence the name.

In his seminal paper in 1971, Lampson [33] formulates the first abstract model of access control from the point of view of operating systems. An *access matrix*, sometimes known as an *access control matrix*, is a two-dimensional matrix with a row for each subject and a column for each object. An element in the matrix specifies the access rights that a subject has on an object. Figure 2.1 is an illustration of an access matrix. An access matrix is a convenient abstraction for expressing discretionary access control polices, and indeed, documents for security requirements of a system often include an access matrix. In a real system, an access matrix would be too large and very sparse. Several mechanisms are available to represent the information in an access matrix. Lampson suggests storing the matrix by rows as *capability lists*, or by columns as *access control lists* (ACL). A capability is a tuple of (object, access rights), and is stored for each subject; an ACL entry, on the other hand, is a tuple of (subject, access rights), and is stored for each object.

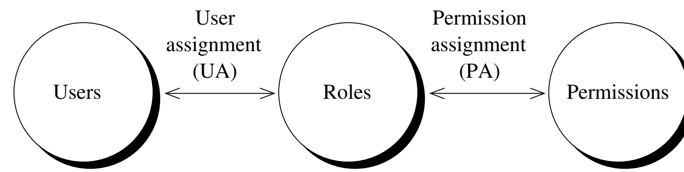


Figure 2.2: A basic RBAC model

As a practical example, the UNIX file system implements discretionary access control. It addresses the size problem of access matrix by effectively reducing the subjects to three (i.e. a 3-row matrix), and represents an access control list by protection bits. The three subjects are: the object owner, group, or everyone in the system. The user who creates an object is the default owner and only the *root* user (i.e. the administrator) can change the ownership of an object. There are three *access modes*: read, write and execute, and the access rights for each subject is represented as a 3-bit value, e.g. `"rwx"`. Every object is associated with a protection string such as `"rwxr-xr-x"`, which in this case indicates the owner has read, write and execute access while the group and everyone else have read and execute access. The key of UNIX access control is that the owner of the object can modify its protection string at his/her own discretion.

2.1.5 Role-based access control (RBAC)

In the 80's, discretionary access control was regarded as suitable for commercial and governmental systems. However, in the beginning of the 90's, the security needs for these systems were more closely examined and it was observed that the protected information was generally not owned by users but rather by the organization or agency to which these users belonged. Moreover, access requests are typically made by a user in the capacity of some role and thus access control decisions are often determined by the acting roles which specify her duties and responsibilities [26, 8]. In the search for a more appropriate access control scheme for civilian systems, role-based access control (RBAC) has gained significant research interest.

The root of RBAC can be traced back to the user grouping found in the UNIX and other operating systems and privilege grouping mechanisms found in some database systems [34, 35]. Over the years, many researchers have proposed models for RBAC [26, 36, 37, 8, 38, 39, 40, 41]. While the differences in these models are quite significant, the core concept remains fairly consistent between them. In RBAC, the basic components are *users*, *permissions*, and *roles*. A user in RBAC typically refers to a human being, although this definition could be extended to include machines, computer processes or autonomous agents. Permissions are defined as an approval to execute an operation on one or more protected objects. An operation could be a simple access mode, e.g. read/write/update, or an complex operation such as a method invocation in an object-oriented system [37, 38]. Indeed, the notion of *abstract permission* exists in early work in operating systems security [33], and RBAC borrows the idea to stress the possibility of high-level operations such as credit or debit for an account in RBAC [42, 8]. The definition of role varies slightly. Some consider a role to be a named collection of permissions [38], while others consider a role to be a job function within the context of an organization [8, 41]. Although both are technically correct, the former focuses on the mathematical definition of a role, whereas the latter emphasizes the use of RBAC in modelling organizational security policy.

Central to RBAC is the notion of relations that connect the components described in the previous paragraph. Suppose U, R and P denote the set of all users, roles and permissions in the system respectively. The *user assignment* (UA) relation is defined as $UA \subseteq U \times R$, which gives a many-to-many mapping from users to roles. Similarly, the *permission assignment* (PA) relation is a many-to-many mapping between permissions and roles, and is defined as $PA \subseteq P \times R$. A schematic illustration of a basic RBAC model is given in Figure 2.2. The arrows represent many-to-many relationships between components.

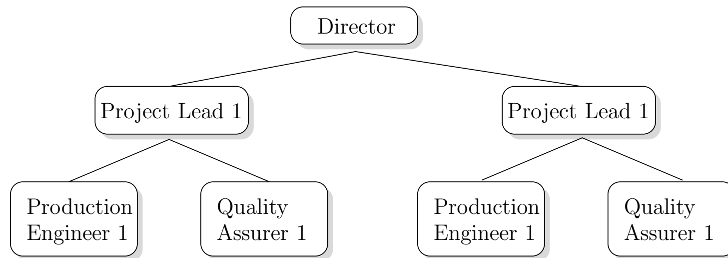


Figure 2.3: An example of role hierarchy

In RBAC, permissions are granted to users only through roles. Suppose a user in a bank attempts to withdraw money from an account, she must be assigned to some role that permits money withdrawal, e.g. cashier. It is possible to assign multiple roles to a single user if the job position demands this. However, it is rare that a user will need all the assigned roles at all times to perform her job functions. The well-known *principle of least privilege* [24] recommends that only those permissions required for a user's context should be available to the user. To address this, many RBAC models [42, 8, 43] incorporate the concept of *sessions*.

A *session* is a one-to-many mapping from a user to roles. A user establishes a session and *activates* some subset of roles that she is assigned to in the context of this session. The permissions available to the user in a session are those assigned to all the active roles in that session. A user may control multiple sessions simultaneously, each acting as a separate instance of the user. The notion of session is analogous to the notion of principal in traditional MAC and DAC, i.e. a session represents an active subject. Since an administrator can restrict a session to only activate needed roles for its designated task, the use of session in RBAC embodies the principle of least privilege.

Another feature commonly found in RBAC models is the concept of *role hierarchy*. The idea behind role hierarchy is due to the observation that roles in an organization can often form a seniority hierarchy, e.g. a Chief Executive Officer (CEO) is more senior than a Vice President (VP). A role hierarchy is closely defined in accordance with this observation, as a partially-ordered seniority relation – see Figure 2.3 for an example of role hierarchy. However, several interpretations for role hierarchy have been proposed. Some researchers [37, 38] favour an permission-inheritance view, whereby role r_1 inherits r_2 if r_1 has all permissions assigned to r_2. Some [26, 43, 39] interpret a role hierarchy in terms of user containment relations, whereby role r_1 contains r_2 if all users assigned to r_1 are also assigned to r_2. Yet others [44] propose interpretation based on role activation, whereby role r_1 inherits r_2 if in all sessions where $r1$ is active, r_2 is also active. In [45], Moffett examines a variety of possible interpretations for role hierarchy. In general, role hierarchy is a structuring tool to model an organization's lines of authority and responsibility. Its main claimed advantage is to improve administrative efficiency by factoring common permissions and reflecting organizational structure.

Advanced RBAC models often offer direct support for expressing conflict of interest policies [24], such as the Chinese Wall policy [25]. This is typically supported through the specification of mutually exclusive roles in *separation of duty* (SoD) relations. Several types of SoD relations have been studied [31, 32, 38, 46, 47]. Static separation of duty relations enforce constraints on the assignment of users to roles, to prevent a user being assigned to two or more conflicting roles at the same time, e.g. a person cannot both be a billing clerk and a bookkeeping clerk. From a policy perspective, while the static constraints of static separation of duty provide a powerful mechanism to prevent mis-administration, it is usually over-restrictive in real-world practice to be useful or even feasible [41], since it is common for a subject to be assigned to conflicting roles especially those in the role hierarchy.

A more relaxed type of SoD relations, known as *dynamic separation of duty*, allows assignment of mutually exclusive roles to the same user but prevents them being activated within the same session. This offers greater operational flexibility in an organization, for example a user can now be assigned with both a billing clerk and an accounts receivable clerk role *provided* these roles are acted on in

independent sessions. Dynamic separation of duty is particularly suitable when sessions are bound with clear and distinctive tasks. Other more complex types of SoD relations include object-based SoD, operational SoD and history-based SoD [31]. The practicality and consequences of these SoD relations remains an open research question, however.

Another aspect of RBAC is its administration. The administration of a RBAC system mainly consists of the specifications of the basic sets, U, R and P and the two relations, UA and PA [38, 48, 39, 41]. In the simplest form, an administrator is allowed to directly create or delete a user, role or permission, and assign or remove a user or permission from a role. An administrator is hence given the maximum power to configure each RBAC component in the system. This monolithic style of administration faces scalability problems for large corporations. A more advanced administrative model, ARBAC97 [48], addresses this problem by applying RBAC to itself. It introduces the concept of *administrative roles*, and encourages partitioning the system into functionally-independent parts which can be separately managed. For example, an organization could have an administrative role for the financial department, responsible for managing the users and roles in the financial department. Likewise, another administrative role could be introduced to take charge of the human-resources department.

One of the most compelling motivations for RBAC is its ease of administration [26, 36, 42, 8]. This is largely due to the additional indirection of roles between users and permissions. Permissions assigned to a role represent organizational security policies, which are relatively constant once established. The administrative task of assigning users to roles, for example when a person newly joins the organization or changes her job position, is considerably easier and less error-prone than directly assigning permissions to each individual. The administrative advantage is particularly important for a large system or an organization with a high turnover of personnel.

Another advantage of RBAC is that it is "policy-neutral". This means traditional policies such as MAC and DAC can be expressed by using role hierarchies and constraints in RBAC [49, 50, 51]. In this regard, RBAC is considered to be a generalized approach to access control. On the other hand, RBAC has some inherent, non-discretionary elements [26, 39]. The roles that a user activates are typically not determined at the user's discretion but rather by her assigned tasks, in compliance with the organizational protection guidelines or security policies, which are usually refined from laws, regulations or operating practices.

2.2 Distributed access control

The work on distributed access control originates from the need to provide authorization on LAN-based distributed systems. Early work includes the Cambridge Distributed Computing System (CDCS) [52, 53], Cambridge File Server (CFS) [54, 55], Hydra [56], and Amoeba [57, 58]. These efforts pioneered the idea of capability-based authorization for distributed systems, which is the predecessor of the modern credential-based authorization. Another thread of the research effort concentrates on providing authentication for distributed systems. The idea is that once a remote user is securely authenticated, access control lists on the server can then be used to provide authorization. Notable work includes the Needham and Schroeder protocol [10] and Kerberos [59, 60].

This section briefly reviews this prior work, and introduces the modern credential-based approach to authorization for large-scale distributed systems.

2.2.1 Access control lists in distributed systems

Some early attempts have been made to reuse the well-known access control lists in distributed systems. The idea is to first authenticate remote users, mapping into local user identifiers, and then rely on the existing access control lists for authorization decisions. With this approach, the security depends heavily on the strength of the authentication scheme.

In their seminal paper, Needham and Schroeder [10] propose the use of cryptographic protocols for achieving secure communications and suggest a key-establishment protocol, based on symmetric

key encryption. A key-establishment protocol allows a shared secret to be established between two principals on different machines and may optionally be used for mutual authentication. The shared secret may subsequently be used for encrypting traffic on the communication channel. The original protocol by Needham and Schroeder has some weaknesses and several suggestions have been described to fix them [61, 62, 63]. Nevertheless, its idea had enormous influence on research in network security, including the well-known Kerberos authentication system.

Kerberos is an authentication service, designed as part of MIT's Project Athena [59, 60], which aims at designing and building an open network computing environment, comprising workstations and various types of servers. The goal of Kerberos is to remove the need for each application to implement its own authentication scheme. Based on the Needham and Schroeder protocol, it adopts a ticket-based approach, whereby a ticket is a server-specific, encrypted token identifying a principal. A ticket is issued by either a Kerberos or a special ticket-granting service (TGS). Prior to making a service request, a client builds another encrypted credential known as an *authenticator* that identifies its name, network address and a timestamp. It then initiates an authentication exchange with the server, passing both the ticket and the authenticator. Once the server decrypts both the ticket and authenticator, and validates their information, it gains confidence in the identity of its communicating party, according to the issuer of the ticket. The original Kerberos protocol is insecure against a variety of attacks [64]. The latest Kerberos, Version 5 [12], developed under the scrutiny of the Internet Engineering Task Force, addresses the known problems and has begun to be widely adopted, e.g. in Distributed Computing Environment (DCE) [65].

2.2.2 Capability-based access control

For distributed systems, one of the inherent problems of access control lists is their scalability limitations. Access control lists can be slow to check, especially if the number of users or groups are large. Moreover, they do not have natural support for delegation – an important mechanism for scaling large-scale distributed systems. With capabilities, on the other hand, access decisions can be made quickly by examining the presented capability. Furthermore, it allows delegation of rights. These considerations make capabilities a more suitable mechanism for distributed systems security.

Early work on distributed system security explores extensively the use of capabilities in providing the authorization need. Notable pioneering work includes the Cambridge Distributed Computing System (CDCS) [52, 53], Cambridge File Server (CFS) [54, 55], Hydra [56], and Amoeba [57, 58]. Capabilities in centralized systems may be protected by hardware (e.g. memory protection). However, in distributed systems where they must be passed around, hardware protection is no longer an option.

One approach is to employ cryptographic techniques to protect capabilities from forgery and tampering. When an object is created, the system associates it with a random secret. The construction of a capability would then involve computing a cryptographic hash of the object identifier, access rights, and the secret. The hash is then embedded in the capability as the check digits. Mathematically,

$$\begin{aligned} hash &= f(secret, protected\ fields) \\ capability &= (protected\ fields, hash) \end{aligned}$$

where f is the hashing function, *secret* is the secret associated with the object, and *protected fields* can include any information, such as the object identifier and the access rights. When a capability is used for access, the system checks if the capability is genuine by recomputing the hash. If and only if the computed hash matches the hash contained in the capability, the system then makes an access control decision based on the capability.

While this approach provides some protection against forgery of capabilities, there are still many issues left unaddressed. For example, it does not detect the use of stolen capabilities, nor does it prevent uncontrolled propagation or duplication of capabilities. Moreover, revocation is often coarsely-grained by resetting the secret of an object (thus invalidating *all* capabilities for the object).

Although the capability-based approach did not provide a complete solution to distributed access control, it was however arguably one of the most important developments leading to today's

access control technologies, with two important implications. Firstly, it experiments with the idea of distributing access rights so that the access control decision at each server can be made simply by validating the credentials presented by a client. Secondly, it prompts the possibility of *privilege delegation* by allowing propagation of capabilities. This effectively decentralizes the task of security management to each client. Both of these implications have impacts on the scalability of a distributed system.

2.2.3 Credential-based access control

With the recognition of the problems in applying capabilities to distributed systems (as discussed in the end of the last section), it gradually became obvious that plain capabilities were unable to satisfy the authorization needs in distributed systems. More information is needed for authorization purposes. *Credentials* are essentially a more elaborate form of data structure, given to and handled by individual principals.

An early form of credential-based access control is due to Li, with his identity-based capability system, ICAP [13]. One important innovation in ICAP is that it merges ideas from both ACLs and capabilities. In ICAP, a capability contains a cryptographic hash computed over the user identifier of its holder, a secret kept by its issuer and the protection information in the capability. This restricts the use of a capability to only its legitimate holder and prevents forgery. It also means that the propagation of a capability must be mediated by its issuer. One novelty in ICAP is its support for selective revocation. A server maintains a data structure called a *propagation tree*, which records the path of capability propagations. If the revocation of a capability is requested by a valid client (i.e. in its propagation path), the server updates its internal secret, thus invalidating capabilities issued with the old secret.

Bull et al. [14], incorporating ideas from [13], describe a credential-based system for Open Distributed Processing (ODP). In ODP, federations of heterogeneous systems are formed, with no central authority nor unified security infrastructure. Considering this level of openness, it becomes obvious that each server is responsible for the management of its own security policy and the enforcement thereof, with a high degree of autonomy. In their design, a server issues access certificates (i.e. credentials) to authorize access to its services. An access certificate is signed with the server's secret key and a client holding an access certificate can freely delegate its access rights to other clients, by adding a signature generated by its own secret key. This process can continue indefinitely and form a *chain of delegation*. On a service request, the server validates the chain presented by a client by recomputing the signatures. Once the validation succeeds, the server applies a local security policy, based partially on the policy identifier contained in the access certificate. An important contribution of this system is the observation that the protocol for authentication could be integrated with delegation, therefore allowing authentication and access control to be performed in a single step. Moreover, its concept of local autonomy and server-oriented security management features an attractive scaling characteristics.

Another credential-based system is described by Neuman [66], in which the author describes the concept of a *restricted proxy*, which is a credential that encodes access rights and use conditions. Similar to previously described systems, it employs cryptographic signing to prevent forgery and tampering. A novel idea in this system is that a proxy includes a set of restrictions that must be satisfied on its use. This allows a principal to delegate a subset of his or her access rights to another principal, achieving fine-grained distribution of access rights. Some restrictions described by the author include a list of issuers, a list of acceptors, group membership, single use, and restrictions on propagation. It also supports the notion of chains of delegation, with the extra flexibility that each intermediary can specify additional restrictions.

OASIS: Open Architecture for Secure, Interworking Services

OASIS (Open Architecture for Secure, Interworking Services) [15, 67] is a recent credential-based access control system, developed at the University of Cambridge Computer Laboratory. It is based on the idea of *principal-specific capability* (e.g. as in ICAP [13]) but is integrated with role-based access

control. While the *protected fields* (see Section 2.2.2) in plain capabilities include primarily an object identifier and access rights, in OASIS, the protected fields are a role name and some parameters for the role (roles are parametrized).

Credentials in OASIS are called *certificates*. There are three types of certificate, *role membership certificate* (RMC), *appointment certificate* (AC), and *revocation certificate* (RVC). A RMC is used to assert a principal's membership of some role, and it can be considered as a transient, session-based capability. An AC is a persistent certificate, designed to implement appointment (which is a more general form of delegation), and a RVC is a certificate to revoke an instance of appointment. In abstract terms, they can be seen as:

$$\begin{aligned} protectedfields &= (\textit{role name}, \textit{parameters}) \\ hash &= f(\textit{secret}, \textit{principal id}, \textit{protected fields}) \\ certificate &= (\textit{protected fields}, \textit{hash}) \end{aligned}$$

An OASIS certificate can only be used by the principal it is issued to. This is achieved by including the principal identifier when computing the hash value. A principal therefore must be authenticated when accessing a service; it is insufficient to simply present a certificate.

A key design feature of OASIS is that it views the system as a collection of *services*. A service may be an *OASIS service* or an *OASIS-aware service*. An OASIS service is in charge of the issuing and revocation of certificates, whereas an OASIS-aware service protects the access of its service by enforcing policies specified in terms of OASIS roles. Services are independently managed and fully autonomous. A service may locally define a set of roles and specify policies governing their use (e.g. the issuing of RMCs, the use of RMCs for service access, etc). Interworking between services is facilitated by *service-level agreements* (SLAs), which specifies the use of RMCs issued by other services. A SLA is typically an agreement between a pair of services, although it is also possible to involve more than two services in a SLA where appropriate.

Another key design in OASIS is its policy-driven nature. The original OASIS includes the Role Definition Language (RDL), which is later refined and formally specified in [3, 5]. Policies can be defined for:

- role activation (issuing of role membership certificates)
- validity for appointment certificates
- service use/method invocation

Policies are based on first-order logic, with the parameter binding semantics comparable to term unification in Prolog. A more detailed description of the OASIS policy model will be provided in Section 6.1.1 when its relation with the Fidelis counterpart is discussed.

2.2.4 Categories of credential-based access control

Most modern distributed access control systems apply the ideas of cryptographically protected credentials as a means of distributing security policies and as a proof of assigned or delegated access rights. The increased adoption of public-key cryptography [68] can also be observed in these systems. This is mainly due to the problem and complexity of key management with symmetric-key cryptography. For distributed access control with this approach, a principal and a service must share a secret key which is distributed online. Moreover, it is desirable to constrain the use of a secret key to each service to limit the damage caused by disclosure of the key. Public-key cryptography significantly simplifies key management because it is sufficient for a communicating party to know only public keys. A credential that binds a public key to some attributes of the holder of the corresponding private key is called a *public key certificate*. This is the most common form of credential found in modern distributed access control systems.

While the general idea of using credentials in provision of distributed access control is widely accepted, the semantics and trust guarantee of credentials differ significantly. Based on how credentials are used, distributed access control may be grouped into two categories:

Identity-oriented approach One common use of access control credentials is to bind the name of a subject with access rights. The idea is that once the name of a requester is proved by a reliable authentication mechanism, access control credentials with the matched name can then be used to make access decisions. This approach separates access control into two distinct stages: authentication and authorization. Authentication requires the binding of a public key to a name, while authorization is handled with the access control credentials which bind a name and a set of authorizations. The security of this approach therefore depends on the reliability of both bindings.

Standards exist for the binding from a public key to a name. Pretty Good Privacy (PGP) [69, 70] and X.509 Public Key Infrastructure (PKIX) [71, 72, 6, 73] are the two most widely used today. The most prominent standard for binding public keys with authorizations is the X.509 Privilege Management Infrastructure (PMI), with its support for attribute certificates [74, 6, 75].

Key-oriented approach Another possible use of access control credentials is to directly bind a public key with authorizations, thus avoiding the use of names completely. With this approach, the public key in an access control credential effectively identifies a subject, and if possession of the corresponding private key can be proved, a service accepting this credential can be sure of the identity of the subject and make access decision simply by examining the access rights in the credential. Unlike the identity-oriented approach, the key-oriented approach integrates the problem of authentication and authorization into one step, but authentication still has to be done.

There are currently two major access control systems based on the key-oriented approach: Simple Public Key Infrastructure (SPKI) [76, 18] by Ellison et al. and KeyNote [19] by Blaze et al.

The remainder of this chapter describes representative systems and technologies for each category introduced above.

2.3 Identity-oriented access control

When Diffie and Hellman introduced public key cryptography as a solution for key management in 1976 [68], they described a “public directory” that lists a user’s name and his or her public key. With this knowledge, one can perform common cryptographic operations such as encryption and signature verification with regard to only the person holding the corresponding private key. While a vast improvement on key management (for secret keys), public key cryptography does not solve the problem but instead shifts the focus to the distribution of public keys. That is, the public directory must be trusted and widely available on demand for this system to be useful and dependable. The public directory is effectively a central point of authority.

Kohnfelder introduced the idea of certificate, or public key certificate in his bachelor’s thesis [77]. The idea is to prevent the possible performance problem caused by a central directory by distributing each entry in the directory as a digitally signed data record, i.e. a data record containing a name and a public key. Since such a certificate is digitally signed, it can be held and passed around by non-trusted parties without violating its integrity. The problem of key management is therefore reduced to the knowledge of the signing authority’s public key.

Public Key Infrastructure (PKI) is a general term to describe the mechanism and architecture to certify the validity and trustworthiness of public-key bindings, traditionally between a name and a public key. Identity-oriented access control extends the use of the name as its premise for access control decisions. This section first examines two popular identity-oriented PKIs and then describes their use in providing distributed access control.

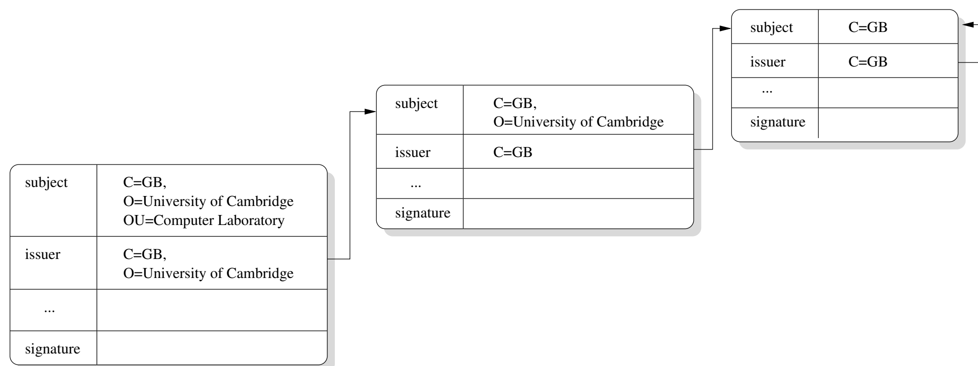


Figure 2.4: A certification path

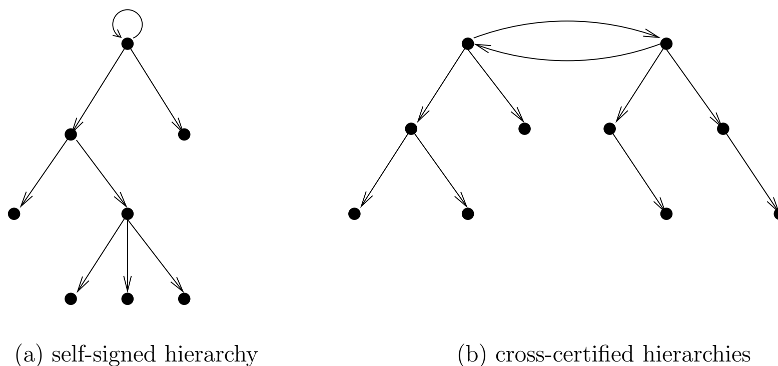
2.3.1 X.509 Public Key Infrastructure

X.509 was originally published in 1988 [71] as part of the X.500 Directory recommendations by the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T), formerly known as CCITT. X.500 was designed to be a global and distributed directory service, whereby an organization can own and administer some portions of the global name space. X.509 was intended to provide authentication and access control for directory entries by binding public keys to X.500 path names (called Distinguished Names). It underwent three major revisions, in 1993 [78], 1996 [72] and 2000 [6], reaching version 3. The version 3 format includes an extension mechanism, allowing binding of arbitrary fields with a public key. The Internet Engineering Task Force (IETF) envisions the need of an authentication framework for secure Internet transactions but since the ITU-T X.509 specification is deliberately over-general, interoperability becomes an issue. To address this, IETF produced an X.509 profile tailored for the Internet, known as PKIX [73], and also a family of protocols for the operation and management of PKIX [79, 80, 81, 82].

The major fields in an X.509 certificate include: a subject name and an issuer name (both are X.500 distinguished names), the subject's public key, a validity period, a version number, a serial number, a digital signature, and a set of extensions. A *Certification Authority* (CA) is a trusted authority that issues, renews and revokes certificates. In X.509 PKI, every CA has a public key certificate to identify itself, and the certificate is signed by a CA with a higher level of authority, and the certificate of that CA may in turn be signed by another CA with an even higher level of authority. CAs are therefore organized into a hierarchical "tree of certification". The original intent of this model is to reflect the design of the X.500 directory service, where there exists a single, global tree of authority, whose root represents the authority of the highest power, e.g. the United Nations. In recognizing the infeasibility of a global tree, IETF's PKIX specifically permits each organization to host its own certification tree to suit its need.

Validation of a public key certificate involves proving the existence of a *certification path*. A certification path is an ordered sequence of certificates, which gives a path from the root of a certification tree to the certificate to be validated. See Figure 2.4 for an example of a certification path. The actual rules for processing a certification path are complex, depending on the extensions used in a certificate. For example, a name constraints extension could specify a list of permitted subtrees in a path. The basic idea of path processing is simple, though: recursively validating each certificate in the path, until a trusted CA, known as a *trust anchor*, or the root CA is found.

There are several *trust models* for the establishment of trust at a trust anchor or a root CA. The IETF PKIX recommends two approaches. A CA may issue and sign a certificate for itself, resulting in a *self-signed* root certificate. Another approach is known as *cross certification*, whereby two or more root CAs or trust anchors issue certificates for each other. Thus, by explicitly trusting one root CA in a hierarchy, one would be able to validate certificates signed by CAs from a different certification



(a) self-signed hierarchy (b) cross-certified hierarchies

Figure 2.5: Examples of trust model

tree. Figure 2.5 illustrates these two common models. A black circle represents a CA, and an arrow represents a certification relationship. Other possible trust models, including bridged hierarchy, trust lists, web-of-trust hierarchies, are described in [83, 84, 85].

Every certificate is issued with a validity period. It states the starting and ending timestamps during which the CA warrants the validity of the information in the certificate. X.509 PKI includes a revocation mechanism for invalidating a certificate before it expires. It models the "blacklist" booklet of bad checking account numbers at supermarket checkouts in the early days. A *Certificate Revocation List* (CRL) lists the serial numbers of revoked certificates. It is created and signed by a CA, and a CA is responsible for periodically publishing its CRL for interested parties.

As noted in the PKIX recommendation [73], one limitation of this style of revocation is that the time granularity of revocation is limited to the issue frequency of CRL. IETF recognizes that where security requirements are critical, online methods of revocation notification will be desirable. Addressing this need, IETF publishes a protocol for checking the certificate status online in 1999 [86]. While this method significantly reduces the delay between the time of revocation and its effect to relying parties, it imposes an extra trust relationship whereby the relying parties must trust the online validation service.

IETF additionally specifies a number of management protocols to support the operations and interactions between a PKI user and management authorities. The main functionalities supported by these management protocols include: registration of a user, initialization of a client system, certification, generation, recovery and update of key-pairs, and revocation request and notification.

2.3.2 Pretty Good Privacy (PGP)

Pretty Good Privacy (PGP) is a software application designed by Zimmermann [69] to allow secure exchanges of files and messages with guarantees of confidentiality, integrity, authentication and to some extent, non-repudiation. PGP is well-known due to its widespread acceptance as a solution for secure e-mail messaging.

PGP is based on both public-key and symmetric-key cryptography. For confidentiality, PGP randomly generates a session key and encrypts the message using a symmetric encryption algorithm with that key. It then encrypts the session key with the recipient's public key and sends both the encrypted message and session key to the recipient as a bundle. For authentication, PGP computes a hash of the message and digitally signs it with the sender's private key. The digital signature is then sent along with the message bundle. It is thus possible to achieve both confidentiality and authentication in PGP by combining both mechanisms.

A user is identified by a name that is, as a de-facto standard, usually qualified with his or her e-mail address. The qualified name is assumed to be unique for the individual's purpose. PGP supports a *web of trust* model, where there is no central authority or hierarchy of authorities for certifying public

key bindings like in X.509. Instead, a name-to-public-key binding is attested by trusted *introducers*, who vouch for the binding by digitally signing it. A user may make any other user he/she trusts an introducer, e.g. based on the past knowledge or personal experience. The underlying theory of this model is that everyone builds up their social circle of trust since their birth, by a large part, judging recommendations and trustworthiness from people they know. PGP believes that by empowering each individual to attest public keys and to accept recommendations from others, one could gradually build up a circle of trust as in the real world.

Each PGP user maintains multiple private and public *key rings*. A private key ring stores the key pairs owned by the user, while a public key ring stores public key bindings the user knows. Multiple public key rings can be maintained to partition their intended use, e.g. businesses or friends. Each public key binding stored in a key ring is associated with a level of trust, a validity score and a list of signatures by its introducers. There are four levels of trust in PGP, namely "unknown", "untrusted", "marginally trusted" or "fully trusted". They are intended to reflect the trustworthiness of the public key owner as an introducer according to the ring owner's knowledge and can be changed by the ring owner at any time. With the trustworthiness information of each introducer and a list of signatures by introducers, PGP computes a validity score for each public key in a public key ring. The validity score of a public key provides a hint to help judge if the key should be trusted.

In PGP, a public key binding is permanent unless the owner of a key ring explicitly removes it. Alternatively, it can be invalidated if a *revocation certificate* exists. A revocation certificate is a negative statement against a public key binding, which prevents PGP from using the named public key. It can only be produced by the owner of the revoked public key, and it is his or her responsibility to distribute the revocation certificate to relevant parties.

2.3.3 Attribute certificates

The concept of *Attribute Certificates* (AC) was introduced by ANSI with the intention to support access control in PKI and was later incorporated into version 3 [72] of ITU-T/ISO X.509 recommendation. In 1999, ANSI published a revision to the original version of attribute certificates, resulting in version 2 [74]. It forms the basis of the work on *Privilege Management Infrastructure* (PMI) in the 2000 edition of X.509 [6], where nearly half of the recommendation is devoted to the subject of PKI-based access control.

The concept of attribute certificates is developed to support authorization in a PKI environment. While it is possible to embed access rights in an X.509 certificate using the extension mechanisms, an X.509 certificate is identity-oriented and its public key binding often tends to be long-lived, therefore ill-suited for expressing authorization. For example, if a person needs to be granted temporary access rights, it would involve revoking the old certificate and issuing a new one. This is not only cumbersome but also conflicts with the idea of identity certification, whereby a public key binding is intended to be stable. Moreover, delegation of rights is often desirable in distributed access control, but again, this notion does not fit well in identity-based certificates.

The idea behind attribute certificates is simple: binding an identity certificate with signed, short-lived certificates that hold *attributes*. There is no constraint on what an attribute can be. For access control, an attribute can be, for example, an access control identity, group/role membership, a security clearance, or other application-specific constraint, e.g. time limit, value limit on a financial transaction, etc. An attribute certificate has a similar structure to an X.509 identity certificate, with the major absence of a subject name. Instead, a *holder* field indicates the linkage to an identity certificate. It can be given as either a general name, a reference to a CA plus a serial number, or a cryptographic hash to be used as the basis for authentication.

The PMI model consists of four components, as shown in Figure 2.6: privilege verifiers, the Source of Authority (SOA), Attribute Authorities (AA), and privilege holders. An attribute certificate is issued and signed by an *Attribute Authority*. Similar to the concept of certification paths, a set of attribute certificates can form a *delegation path*. The root of a delegation path is called the *Source of Authority*, and is trusted with the management of authorization for the whole system. It delegates a partial management responsibility to an AA by issuing attribute certificates with special delegation

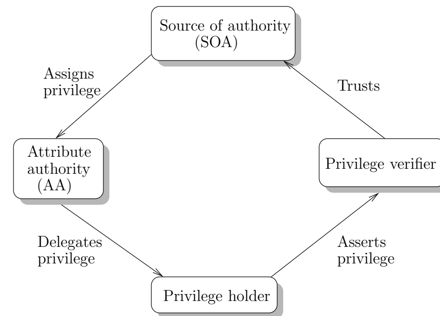


Figure 2.6: PMI delegation model (simplified from [6])

extensions. That AA could further delegate to other AAs or end users. An attribute certificate for an end user effectively means delegation of access rights from the issuing AA to the user. One requirement of delegation is that an AA participating in a delegation path could only delegate access rights, or a subset of them, that have been issued to it, i.e. delegation is *monotonic*. Delegation in X.509 PMI could be subject to various restrictions, using delegation constraints. For example, a pathLenConstraint extension specifies the maximum allowed distance between an issuer and a privilege verifier.

An AA is typically a separate entity from a CA. While some CA may incorporate the functionalities of an AA, it cannot be assumed that a CA will possess sufficient knowledge to determine authorization for its users in general. In this model, a privilege verifier trusts the SOA to delegate its access rights to AAs or end-user privilege holders; it trusts the SOA as the authority over the control of protected resources. This model separates the administration of access control policies from their enforcement.

X.509 PMI optionally supports role-based access control (RBAC) through the use of extension fields. An attribute certificate with the role specification extension is called a *role specification* certificate, which associates a role name with a set of access rights delegated by an AA. An AA may also issue *role assignment* certificates that associate individuals with roles. A privilege verifier presented with a role assignment certificate derives the access rights of a privilege holder by asserting the role specification certificate of that role, which may be known beforehand, together presented by the privilege holder, or discovered by some other mechanisms.

Revocation of attribute certificates is supported using the same Certificate Revocation List (CRL) mechanism as in the X.509 PKI. In addition, PMI defines two extension fields for use in an attribute certificate to assist revocation. A CRLDistributionPoints extension instructs a privilege verifier to fetch the CRL from the specified location. An attribute certificate could also contain a NoRevAvail extension, which informs a privilege verifier that no revocation will be made on this certificate. This may be useful in some situations, e.g. certificates with a very short validity period, thus revocation checking may be omitted for efficiency.

Access control systems based on and/or extending the use of attribute certificates include: Akenti [87, 88], Globus [89, 90] and PERMIS [91].

2.4 Key-oriented access control

Identity-oriented access control is centred around the concept of *names*, which are intended to be associated with both access rights and the real principals. However, the very notion of names becomes problematic in meeting the access control needs of large-scale, widely distributed systems. First, the identity-oriented approach assumes that a name uniquely identifies a principal. In a large distributed system, a global naming scheme imposes several problems, e.g. scalability, flexibility. Moreover, a naming scheme usually has a fixed (hierarchical) structure. It is impossible to devise a single structure

that satisfies the need for every application. Second, names do not contribute much in deciding access when there is no prior experience or relationship between a service requester and a provider. In the real world, one function of names is to link relevant trust information regarding an entity together. Based on the knowledge of the trustworthiness of an entity, a service, e.g. a bank, can then make an informed judgement as to whether to provide access or not.

Despite the relatively few use of names, the concept of globally unique identifiers is nevertheless essential for access control; a computer system still needs some means to reliably identify its requester. The basic idea of key-oriented access control is to use public keys as principal identifiers. The assumption is that every principal generates their own key pair and is responsible for safeguarding their private key. By requirement, a public key generation process must produce globally unique keys, otherwise the public key cryptosystem is considered flawed. Since a private key is kept secret, presenting a public key to a service and proving the knowledge of its corresponding private key is sufficient as a proof of the owner of the key pair. Public keys thus qualify as globally unique identifiers for the purpose of access control.

This key-centric view removes the dependency on names, which means no naming authorities are required in the system. An important implication is that the trust relationship is simplified because a service does not need to explicitly trust the assertion of a name binding (i.e. authentication) by a naming authority. A service is responsible for authenticating its own requesters and deciding their access rights. The key-oriented approach effectively blurs the distinctions between the two phases of access control, authentication and authorization. By allowing full control over their own resources, the key-oriented approach offers more service autonomy than the identity-oriented one.

This section briefly surveys some of the recent key-oriented access control systems and *trust management systems*, which extend the concept of key-oriented access control with the management of trust relationship and security policies.

2.4.1 Simple Public Key Infrastructure (SPKI)

Simple Public Key Infrastructure (SPKI) is a work-in-progress standard by the IETF SPKI Working Group, tasked with producing a certificate structure and operational protocols to support the needs of authorization management in Internet applications. This work was originally motivated by the inflexibility and inadequacy of the global naming hierarchy in X.509. Separately proposed, Simple Distributed Security Infrastructure (SDSI) by Rivest and Lampson [17] was also designed to address the same global naming problem. The two projects were later merged and published as the SPKI/SDSI 2.0 [76, 18, 92].

In SPKI, the view of the world is fully key-centric. Every principal, including a person, a process, or a service, may freely generate a cryptographic key pair and is identified by their public key. Every principal can sign and issue certificates using their own private key, and a signed certificate can be verified by any principals with the public key of the signer. There are three types of certificate in SPKI: *authorization certificate*, *name certificate* and *access control list (ACL)*. An authorization certificate is the most common type of certificate, and sometimes is just called a "SPKI certificate" or simply "certificate". It transfers some specific access rights from one principal to another, i.e. it is a delegation certificate. A name certificate binds a public key with a name. SPKI supports the SDSI linked name model, described later. An access control list is a special type of certificate that represents the security policy of a service. It is not intended to be distributed, but rather held in secure storage private to a service.

Authorization certificates can form chains where access rights are delegated from one public key to another. When a service, S, grants access rights to a principal, C_1, it issues an authorization certificate that carries delegated rights from its ACL. C_1 could issue and sign an authorization certificate to further propagate this delegation to C_2, and so on. When S is requested by C_n, a certificate with the delegated access rights is presented to the service, completing an *authorization loop*, illustrated below. A double arrow indicates a delegation and a single arrow indicates a service request.

$$S \Rightarrow C_1 \Rightarrow C_2 \Rightarrow ... \Rightarrow C_n \rightarrow S$$

Validation of requests in SPKI is based on a technique known as *tuple reduction.* The idea is to complete an authorization loop, given a chain of certificates. Authorization certificates in SPKI are represented as a 5-tuple:

$$(I, S, D, A, V)$$

where I is the issuer's public key, S is the holder's public key, D is a Boolean indicating whether further delegation is permitted, A is a set of access rights, and V is the validity period. For S, one can specify a k-of-n *threshold subject* to indicate that k out of n subjects must sign to validate the delegation. Access rights are defined using *tags*, whose interpretation is left to an application. The tuple reduction reduces two tuples $(I_1, S_1, D_1, A_1, V_1)$ and $(I_2, S_2, D_2, A_2, V_2)$ into $(I_1, S_2, D_2, A_1 \cap A_2, V_1 \cap V_2)$, provided all the following are satisfied,

1. $S_1 = I_2$
2. The two intersections succeed,
3. $D_1 =$ **true**.

The intersection for access rights derives the most restricted authorizations between the two tuples. Although access rights are defined in an application-dependent manner, SPKI attempts to define rules to allow automatic processing. The intersection between two validity periods computes the overlapping period between two, or fails if the two periods do not overlap.

A SPKI certificate has validity dates that give the lower and upper bound of its validity period. It is also allowed to validate using online methods, including the X.509-style CRL, timed CRL, online status query, timed revalidation and one-time revalidation. There are other possibilities being considered, and it is still an open area of discussion at the time of writing.

In SPKI, a concept of local names is supported to give a binding from a key to a human-recognizable name. Local names are defined within the local namespace of a principal, similar to names in a personal address book. Local names do not need to be globally unique, but need to be unique local to the principal who defines them. A globally unique version of a name could be obtained by linking a local name with its namespace, resulting a *linked name.* This is similar to say "the person known as John Smith by the University of Cambridge". For example,

```
fred: (name sam)
```

defines a principal named `sam` known by `fred`. Another principal `george` could refer to the same principal in terms of the knowledge of `fred` by

```
george: (name fred sam)
```

Name certificates are represented in SPKI as a 4-tuple:

$$(I, N, S, V)$$

where I is the issuer's public key, N is a name given as a byte string, S is the holder's public key, and V is a validity period. There are two classes of 4-tuples, those that define a name for a public key and those that define a name as another name. Tuple reduction rules for 4-tuples concatenate a chain of 4-tuples into a public key. Depending on the classes of a name definition, on every step, a name is either resolved into a public key or a reference to another name. To avoid naming loops, SPKI requires either chains of names to be provided in order, or when an unordered pool of tuples is supplied, that only those names with a binding to a public key will be processed.

2.4.2 PolicyMaker and KeyNote

The concept of *trust management* was first introduced by Blaze et al. [9], who define it as "a unified approach to specifying and interpreting security policies, credentials and relationships that allows direct authorization of security-critical actions" [16]. At the heart of a trust management system is a set of general-purpose mechanisms for handling security policies and credentials, and deciding policy compliance. They developed PolicyMaker as a proof-of-concept trust management system and demonstrated its use in several applications, including medical applications [93], network protocols [94, 95], and Internet content rating applications [96]. Building on the experience of PolicyMaker, they developed its successor, KeyNote, and published it as an IETF Request for Comment (RFC) [19, 20].

The PolicyMaker system centres around a *trust management engine*, which is essentially a query engine that evaluates a requested action against local security policies. The trust management engine takes as input: an action string, the local policies, and credentials presented by the requester. The response to a query could either be a simple yes/no result, or additional restrictions that would make the requested action consistent with the local policies. A query to the PolicyMaker engine has the following syntax:

$$key_1, key_2, ..., key_n \textbf{ REQUESTS } \textit{ActionString}$$

An action string is an application-defined description of an action requested by one or more principals, identified by their public keys. Its semantics is only of concern to the application and the trust management engine does not depend on it. Both policies and credentials are referred to as *assertions*. An assertion is essentially a construct that delegates authorizations to perform actions to a principal from its signer. An assertion has the syntax:

Source **ASSERTS** *AuthorityStruct* **WHERE** *Filter*

Source is the source of the assertion, which can either be the keyword **POLICY** in the case of policy assertions or a public key of the principal who confers the authority implied by the assertion in the case of credentials. *AuthorityStruct* specifies a list of principals to whom the assertion applies. Each principal could be specified as a single public key, or as a threshold structure, i.e. k-out-of-n keys. *Filter* specifies the conditions that an action string must satisfy for the assertion to be valid. Filters are in fact, by design, interpreted programs. This allows maximum power and expressiveness. However, the absolute power of filters poses security concerns. It is therefore required for the filter programs to be executed in a "safe" sandbox or implemented in a safe language. The PolicyMaker prototype is equipped with three filter languages: AWKWARD, which is a safe version of AWK, Java and Safe-TCL.

While both credentials and policies share the same assertion syntax, they differ in one significant respect: credentials are signed assertions, whereas policies are not signed. Credentials are intended to exist outside the trust management engine and therefore must be signed to protect their integrity. Policies, on the other hand, are purely for local use and are unconditionally trusted by the trust management engine. Signing policies is hence unnecessary. The set of policy assertions on a system forms a *trust root*. Analogous to SOA in X.509 PMI, they are the ultimate source of authority for the trust decision about a request.

The processing of a query is called the *proof of compliance*. The compliance checking algorithm in PolicyMaker is formally specified and analyzed in [97]. In essence, the algorithm attempts to find a chain of delegation from some trust root to the public keys requesting the action in which all filters along the chain are satisfied. Filters take as input the current action string and an environment, which contains information relevant to the evaluation context, e.g. date, time, etc. Filters also have access to assertions in the chain being evaluated. An application designer is thus empowered with the ability to express filters that enforce contextual constraints such as expiration times, or limit the degree of delegation.

An assertion in a chain may modify the current action string through the use of *annotations*. The annotation mechanism is designed for inter-assertion communications, where the outcome of an evaluated filter in an assertion may influence the evaluation of the filter in the next assertion in the chain. This enables an assertion to append additional conditions to an action string, if the policy requires it.

As a motivating example for the PolicyMaker compliance checker, consider a policy where an online entertainment company allows streaming contents to be delivered to its customers. A customer is certified by the company's customer service department, with the public key `customer_dept_key`, based on criteria such as the payment of subscription fee, type of subscribed services, etc. The policy may be expressed as follows:

```
POLICY
ASSERTS customer_dept_key
WHERE a filter that allows streaming video for a "customer" role
```

The customer service department issues and signs credentials to customers, stating the owner of a public key is a valid customer. An example credential for a customer Alice whose public key is `alice_key` is given here.

```
customer_dept_key
ASSERTS alice_key
WHERE a filter that returns true if role is "customer"
```

When Alice wishes to view an online streaming concert, she needs to present the credential assertion to the streaming server, which composes the following query for the PolicyMaker engine:

```
alice_key REQUESTS "streaming video" in the capacity of a "customer"
```

This query results in an acceptance, because the PolicyMaker engine is able to find a chain consisting of the trust root (i.e. the policy) and a credential assertion by the customer service department satisfies the requested action.

KeyNote is based on the same concepts as PolicyMaker but with additional emphasis on standardization and ease of integration into applications. This is largely built on the experience of PolicyMaker, where the freedom of the filter languages presents obstacles for interoperability. Furthermore, applications are required to perform cryptographic verifications against credential assertions, which complicates integration. Addressing these issues, KeyNote provides a single, unified assertion language, which is designed to work smoothly with its compliance checker. It also shifts more responsibilities from applications to the trust management engine, e.g. signature verification.

Similar to PolicyMaker, the KeyNote trust engine takes a list of credentials, policies, public keys identifying requesters and requested actions in a query. Actions are specified as a collection of name-value pairs, called an *Action Environment*. The values are application-specific, and it is the responsibility of the calling application to construct and gather all information needed to evaluate a trust decision. The KeyNote engine returns a *policy compliance value* as a result of a query. The policy compliance value is configured by applications, and is intended to provide the calling application with more information on how to proceed with a request. In the simplest form, this is a Boolean result, e.g. accept or reject.

KeyNote defines a human-readable format for its policies and credentials, based on RFC822-style e-mail headers. A credential from the KeyNote RFC [20] is given in Figure 2.7 as an example. The Authorizer field is mandatory in all assertions. It identifies the issuer of an assertion. For policy assertions, this must be the keyword **POLICY**. The Licensees field identifies one or more principals authorized by the assertion. For example, Figure 2.7 restricts the use of the assertion to any of the named public keys. The Conditions field is essentially a highly-structured program that tests action environments. KeyNote provides string comparisons, numerical operations and comparisons, and regular-expression comparisons.

```
KeyNote-Version: "2"
Authorizer: "DSA:4401ff92"   # the Alice CA
Licensees: "DSA:abc991" ||   # jf's DSA key
           "RSA:cde773" ||   # jf's RSA key
           "BFIK:fd091a"     # jf's BFIK key
Conditions: ((app_domain == "RFC822-EMAIL") &&
             (name == "J. Feigenbaum" || name == "") &&
             (address == "jf@keynote.research.att.com"));
Signature: "DSA-SHA1:8912aa"
```

Figure 2.7: Sample KeyNote assertion

The compliance checking model of KeyNote is a subset of PolicyMaker's. It employs a depth-first search that recursively attempts to satisfy at least one policy assertion. Satisfaction of an assertion requires both the Conditions and Licensees fields to be satisfied. It is claimed by its designers that the simpler compliance checking algorithm in KeyNote, while more restrictive, is more efficient than the one in PolicyMaker.

A last note on both PolicyMaker and KeyNote. Both systems are *assertion monotonic*, i.e. negative assertions against principals cannot be specified. This is regarded by their designers as a higher-level feature that should be provided by applications if required.

2.4.3 Other trust management systems

REFEREE [98, 99], which stands for Rule-controlled Environment For Evaluation of Rules and Everything Else, is a trust management system designed specifically for the World Wide Web. Its primary goal is to help users decide what to trust on the web. It is based on similar ideas developed in PolicyMaker, including recommendation-based trust and fully programmable credentials and policies. It nevertheless differs significantly from PolicyMaker in several respects:

- The REFEREE trust management engine is able to fetch additional credentials to assist policy evaluation during its execution. This is considered useful in the web setting, where, for example, a user may wish to obtain a particular reviewer's opinion about a video clip before downloading it.
- It supports non-monotonic assertions. Policies and credentials may be used to express denial of specific actions. This is consistent with the notion of parental control over web content viewed by their children.
- It employs a fully policy-driven approach. Both its policy evaluation and credential fetching mechanism are directed by policies. A REFEREE policy is essentially a program that not only filters attributes but is also allowed to download and invoke other REFEREE programs.

There are three primitive data types in REFEREE: tri-values, statement lists and programs. A tri-value is either true for acceptance, false for denial, or unknown if there is insufficient information to either accept or deny. A statement list is a set of assertions, expressed in two-element s-expressions, similar to name-value pairs but also allows nesting. For example, an assertion stating a web page is signed to have been virus-checked would be:

```
("code-signing" ("virus-checked" TRUE))
```

Both policies and credentials are programs that take a statement list and return a tri-value. The differences lie in the intention of a program. A policy infers the compliance of a given statement list and the tri-value indicates the result. The optional statement list may be returned as a justification

of the decision. A credential, on the other hand, introduces new assertions based on the input and the returned tri-value is merely an indication of the state of execution, e.g. successful or failed.

A query to the REFEREE trust engine takes a policy name and additional arguments, including credentials or statement lists. REFEREE then downloads the relevant policies and executes them. A policy may recursively download and invoke other policies until the execution terminates and a tri-value and an optional statement list are returned.

Programs in REFEREE are coded in *Profile-0.92*, which is a policy specification language designed to work with the W3C PICS (Platform for Internet Content Selection) [100] labels. It offers a label-loading subroutine, tri-value operators and pattern matching operators on statement lists. Each rule is an s-expression, with the first element being an operation, followed by arguments. As an example, a sample policy is given here that asserts true if and only if a web page is rated by the Guardian system as suitable for anyone (i.e. the minimum age is below 12).

```
(invoke "load-label" STATEMENT-LIST
                     URL "http://www.guardian.org/" (EMBEDDED))
(false-if-unknown
  (match
    (("load-label" *)
     (* ((version "PICS-1.1") *
         (service "http://www.guardian.org/") *
         (ratings (RESTRICT < minimum-age 12)))))
   STATEMENT-LIST))
```

IBM Trust Establishment (TE) [101] is a trust management system that features role-based access control. Similar to many other systems, it uses public keys as principal identifiers. The central component is the Trust Policy Language (TPL), which is an XML-based language that maps credentials (held by a principal) into roles. Roles are treated as groups of principals that represent specific organizational units, and their memberships depend on rules specified in TPL. A TPL rule defines the set of necessary credentials and conditions on their fields for joining a role. As an example, a policy for an online chat room that states a user can become a chat room member if recommended by two existing members could be specified as below.

```
<POLICY>
  <GROUP NAME="members">
    <RULE>
      <INCLUSION TYPE="Recommendation" FROM="members" REPEAT="2" />
    </RULE>
  </GROUP>
</POLICY>
```

IBM TE is designed to be independent from credential formats. A credential framework that maps a variety of formats, including X.509, SPKI and KeyNote, into a generic credential structure is described in [102]. The mapping process involves translating different encodings into a common interface and then resolving semantic differences. Generic credentials are statements signed by an issuer, identifying properties of a subject and its public key. In addition to the public keys of the subject and issuer, a credential also contains a type, addresses of credential repositories, and a profile identifier. A credential type identifies a credential profile, which defines the syntax and semantics of the credential. Both the issuer and the subject could manage their own credential repository. The issuer's repository is intended for checking credential revocation and for listing "black-listed", negative credentials. The subject's repository, on the other hand, is intended to allow the trust management engine to locate and collect missing credentials automatically.

The *RT* framework [103] is a new trust management framework that integrates concepts from role-based access control. It includes a family of languages for expressing policies and credentials, RT_0, RT_1, RT_2, RT^T and RT^D. RT_0 is the base language that supports conditional assignment of

principals to roles. RT_1 extends RT_0 with parametrized roles. RT_2 adds to RT_1 a notion called *logical objects*, which are grouping of objects, similar to the way a role groups principals. RT^T and RT^D can be used on top of RT_0, RT_1 or RT_2. RT^T adds constructs for expressing threshold principals and separation of duties, whereas RT^D adds delegation of role activations.

A RT credential has two parts: the head and the body, where the head is a role name and the body is a list of conditions for becoming a member of the role. It essentially represents a single logic rule. As an example, a type of RT_1 credential has the syntax:

$$A.r(h_1, ..., h_n) \longleftarrow B.r_1(s_1, ..., s_m)$$

where A and B are principals, r and r_1 are role names, h_i for $1 \leq i \leq n$ and s_j for $1 \leq j \leq m$ are parameters. This credential means that provided a principal is a member of the role $R_1 = r_1(s_1, ..., s_m)$, defined by B, then it will also be granted the membership of the role $R = r(h_1, ..., h_n)$, defined by A. In addition to this type of credential, there exists three other types of credentials:

- $A.r(h_1, ..., h_n) \longleftarrow B$: directly assigning principal B to the role $R = r(h_1, ..., h_n)$.
- $A.r(h_1, ..., h_n) \longleftarrow A.r_1(t_1, ..., t_l).r_2(s_1, ..., s_m)$: this is the so-called *attribute-based delegation*, where A assigns the role $R = r(h_1, ..., h_n)$ to any principal who is granted the role $R_2 = r_2(s_1, ..., s_m)$ by a principal who is in the role $R_1 = r_1(t_1, ..., t_l)$.
- $A.R \longleftarrow B_1.R_1 \cap ... \cap B_k.R_k$: A assigns the role R to any principal who is in the role $B_i.R_i$ (defined by B_i), for $1 \leq i \leq k$.

The main novelty of the RT framework is its tight integration with the notions from role-based access control, including hierarchical roles, role-based separation of duty, and role-based delegation. This is evident in their special treatment of credentials. With the rule-based approach, credentials can essentially be considered as pre-written policies, designed to facilitate RBAC for decentralized environments.

2.5 Summary

This chapter has presented an overview of the research in security policies and access control models. It has also briefly reviewed the research on distributed access control, starting from the early days of capability-based systems, to the modern credential-based systems. Two kinds of approaches to credential-based access control have been described. The identity-oriented approach uses credentials as assertions for the binding of a public key with a name. The representative work in this area is the ITU/ISO X.509 Public Key Infrastructure (PKI) and Privilege Management Infrastructure (PMI). The newer, key-oriented approach associates a public key directly with access rights, thus avoiding the use of names. Simple Public Key Infrastructure (SPKI) and PolicyMaker/KeyNote are the representative work in this area. In particular, PolicyMaker/KeyNote, with an integrated approach to the specification of security policies and trust relationships, are known as *trust management systems.*

3 Fidelis Trust Management Infrastructure

This chapter introduces Fidelis, a decentralized trust management framework. It begins with an overview of the Fidelis trust management infrastructure, outlining the basic concepts and key features. Section 3.2 presents a review of the concept of trust in the literature, from both the computer science and sociology perspectives. It finishes with a discussion on the common factors that influence one's trust decisions. These factors serve as a basis for Fidelis. Section 3.3 describes the *trust conveyance model*, which attempts to model mechanisms that propagate trust information in daily life, and discusses the rationale behind this approach. In Section 3.4, the key-centric approach of Fidelis is described and its appropriateness for the conveyance model is discussed. Section 3.5 describes the *Fidelis Policy Language* (FPL) – a language for specifying trust-related policies. This section ends with a discussion comparing the Fidelis Policy Language with similar existing research.

3.1 Overview of the Fidelis Trust Management Infrastructure

Fidelis is a framework for specifying, expressing, and managing trust information for Internet-scale distributed applications. In Fidelis, a *principal* may be a person, an organization, a computer process, or any other entity in some authority. A principal is identified by public keys, which it can generate at any time. The world is considered as a flat space, in which every principal may potentially interact with any other. Local structures, however, may exist to promote better manageability, e.g. an organization may form a hierarchy reflecting its internal structure, but externally it may be identified as a single organizational principal.

Fidelis is based on the *trust conveyance model*, whereby a principal may freely pass beliefs or assertions to others. These are modelled as *trust statements.* An instance of a trust statement (called a *trust instance*) has an explicit *truster* and *subject.* It is represented as a public key credential, signed by the truster. It has a validity condition, which is defined by the truster to enable invalidation of outdated trust beliefs.

A principal may specify *policies* relating to trust statements. A principal may describe its policies by any convenient means, and this usually depends on the complexity and scope of policies. Nevertheless, a language is designed to be a reference for policy specification, called the Fidelis Policy Language. In the Fidelis Policy Language, trust statements are represented as predicates with typed parameters. The parameters are intended to expose details about a trust statement instance, which refines the granularity of policies. It includes the notion of *action* to model requests. An action is an abstraction that may correspond to a method invocation, a service request, or other behaviour that may be subject to trust decisions.

The language allows two types of policy: *trust policies* and *action policies.* A trust policy defines a trust relationship, while an action policy relates action and trust. A trust policy is defined in terms of *prerequisite* trust instances, and may include conditions on the parameters of these trust instances. It may also define a "blacklist" of trust instances, which must not exist for a trust relationship to be formed. An action policy has a similar structure but is specified for actions.

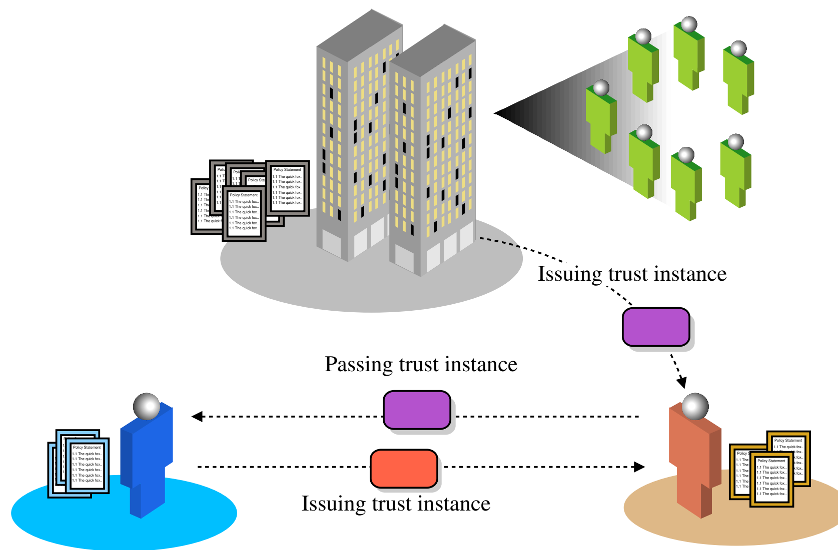


Figure 3.1: Fidelis overview

Using this language, a principal may query a *policy engine* for: (1) whether a new trust relationship can be formed, (2) whether new trust instances meet specific requirements, (3) whether an action complies with the local policies, or (4) what actions comply with the policies. The policy engine, depending on the types of queries, returns either a Boolean result or new trust or action instances, and may additionally include an execution trace, which gives justifications for the decision. A principal may further interpret the returned trust or action instance in the application context to act upon the trust decision.

Figure 3.1 gives an overview for the Fidelis trust management framework. Every principal defines its own policies, and may create new trust instances and exchange trust instances it knows. The organization in the figure functions like other principals and is also identified by a public key. There may be many principals within the organization. They are also like ordinary principals and can convey trust, but not on behalf of the organization to which they belong.

3.2 Trust model

The concept of *trust* occurs in many branches of computer science. However, because of its abstract and elusive nature, there is a tendency in the literature to tailor the meaning of trust to its specific use in a particular application domain. For example, as in the terminology of classic security of *Trusted Computing Base (TCB)*, *trust* means compliance with the security policies under the assumption of correct functioning of hardware; in authentication protocols, *trust* may refer to the safe and secure handling of secret keys by a key distribution centre. In e-commerce, *trust* may relate to the fulfillment of payment and/or delivery of goods.

Trust is a very general topic that may be applied to virtually any context. The lack of a consensus definition of trust reflects its complexity and generality. This section describes the notion of trust on which Fidelis is built. It does not attempt to define a unified trust model but instead proposes a framework in which different trust models might coexist. Towards this goal, it is essential to understand different meanings of the term “trust”. In fact, various interpretations exist not only in computer science but in other classic sciences where trust has been widely studied, including politics, psychology and sociology. It is therefore important to consider these disciplines also in order to capture the essence of trust. We will however first discuss trust in computer science, specifically in security research.

3.2.1 Trust as a security concept

Trust is one of the most important foundations of information security. The basis of security relies on the correct operations of hardware and software, the correctness of cryptographic algorithms, the correctness of cryptographic protocols, etc. Even without being explicitly stated, trust is placed on every link in the chain of security for a system to be considered *trusted.* If any of the components in a link were broken, the security of the system would be defeated. In this regard, the concept of computer security is tightly related to *dependable computing*, wherein the notion of *trust* has an element of *reliance* in both areas. The United States Department of Defense Trusted Computer System Evaluation Criteria (TCSEC) [104] is the earliest *trust assurance* policy, designed for military systems. The idea is to evaluate a computer system against a set of formally specified criteria to determine its level of trust. Trust in this sense is equivalent to dependability. It is a positive belief that the system will operate with a certain level of confidence, reliability and dependability [105, 106].

Trust can also be observed in cryptographic protocols although somewhat implicitly. For example, the basic idea of authentication protocols is to derive a specific type of trust as a conclusion: the belief that the communicating entity is indeed the claimed principal. Depending on the details, the execution of a protocol often needs to make a number of trust assumptions on either end of the communication, e.g. the belief that the server will generate a session key of a sufficient strength, the belief that the server will not leak out confidential information, etc. The observation here is that trust is relative to specific tasks [107, 108]. Trusting a server for authentication does not imply that the server should be trusted for secure storage of confidential data. The *purpose* associated with trust must be explicitly stated. Based on this idea, Yahalom et al. define trust as a belief that a principal has the potential to complete the specified tasks competently and honestly [107].

Explicit applications of trust in security can be found in formal logic [109, 110, 11]. Burrows et al. developed a logic (referred to as the BAN logic) for the verification of authentication protocols [109]. It introduces connectives for expressing beliefs ($\models$) and jurisdiction ($\Rightarrow$). Its jurisdiction rule states that if P believes that Q has jurisdiction over fact X and Q believes X, then P believes X. This is written as a sequent,

$$\frac{P \models (Q \Rightarrow X), P \models Q \models X}{P \models X}$$

If A explicitly trusts a certification authority S for providing B's public key, this relationship can be expressed as a jurisdiction,

$$A \models (S \Rightarrow \stackrel{K_B}{\mapsto} B)$$

where $\stackrel{K_B}{\mapsto} B$ means B has public key K_B. If $S \models \stackrel{K_B}{\mapsto} B$, then by the jurisdiction rule, we can conclude that $A \models \stackrel{K_B}{\mapsto} B$. The jurisdiction rule defines a trust relationship on the basis of *belief* and *truth.* This approach, while specific to its domain, is an appropriate definition for trust because there is an absolute notion of truth in cryptographic protocols, e.g. the fact that a principal owns a key can be verified by encryption/decryption of a secret. Trust in BAN is the belief that a given principal has authority over the truth of a fact.

Another form of trust can be seen in the logic for distributed authentication by Lampson et al. [11], which includes a construct for describing delegation of rights. They defined a *speaks for* relation ($\Rightarrow$), where $A \Rightarrow B$ means that if A says any statement, we can believe that B says the same statement. This type of trust encompasses the notion of *honesty.* If principal A is trusted to speak for B, then it is believed that A will honestly say a statement that B also says. It is noted in [111] that as well as honesty, the concept of *responsibility* should also be considered in delegation. Responsibility is a means of managing *risks* so that, for example, the possible damage and liability of an action by a delegated principal can be accounted for. This crucial observation suggests that trust has an intimate connection with risks. Indeed, this shares the view with social and psychological aspects of trust, which will be discussed later.

The trust relationship expressed by the *speaks for* relation exhibits a strong, context-less belief. When $A \Rightarrow B$ is trusted, then *every* statement made by A is believed to be also made by B. [11] includes the concept of *roles* to allow a principal to limit its authority. A role may be defined as the name of a program, e.g. NFS server, or its class, e.g. untrusted file server. Principal A acting in role R is written as A **as** R. A weaker trust relationship of *speaks for* can be expressed as $A \Rightarrow (B \textbf{ as } R)$. This means if A says some statement, it is believed that B as role R says the same statement. While this approach is somewhat cumbersome to limit the scope of trust, it recognizes the importance of making trust more specific, which corresponds to the concept of *trust purposes* discussed earlier.

Another significant modelling of trust can be seen in public key management, where the term *trust model* is used to describe the structure of certification authorities, recognizing that the monolithic, single-tree approach of the original X.500 is unlikely to be realized. The literature [84, 85, 112] has suggested a number of structures, e.g. strict hierarchy, cross-certified hierarchy, bridged hierarchy, etc. The concept of "trust" in these work is narrow, referring specifically to the *authority to certify keys*. The use of the term "trust model" here could in fact be more precisely described as *certification topologies* [113].

As reviewed in this section, trust in security assumes complete certainty. If a computer system is certified to be trusted at a certain evaluation level, it implies it should always function within the guarantees of that level provided correct operating procedures are followed. In logic, if a principal is trusted, it means it will always demonstrate certain expected properties, e.g. to have jurisdiction on asserting public keys for some principals. Trust in security research is taken as a binary concept. It makes little sense to say a certification authority guarantees a public key 80% of the time, or an evaluation criteria to guarantee 65% of the operational time of a system. Jøsang [114, 108] describes this type of trust in his model as a belief by *rational entities*, which are defined as entities that will resist malicious attacks. This is opposed to *passionate entities*, which are entities with free will and possess human-like behaviour. Classic sciences such as psychology and sociology provide a wealth of study on trust in human societies, and thus help us understand passionate entities in computer environments.

3.2.2 Trust as a sociological concept

In contrast with those somewhat simplistic views of trust adopted in security research, trust has been studied in a much wider context in other disciplines. Generally speaking, the word "trust" is often used by people in a very broad sense to mean a number of things. Its interpretation by the trusting party varies significantly, depending on past experiences [114, 115, 116], associated risks [116, 117, 118], recommendations from other parties [114, 119, 120], reputation of the trusted parties [121, 122, 123], or even cultural background [124, 122]. It is not always clear to every person how trust or distrust is derived in every case, and indeed, sometimes this process occurs subconsciously. For example, some people base their trust decisions strongly on *first instinct*, or psychologically place more trust on people of their own race. However, there is a fairly uniform recognition among researchers that trust is a *subjective* measure [125, 123, 126, 85]. Given the same external conditions, people may often have a different degree of trust over the same matter. This is illustrated by Gambetta's definition of trust [125]:

> "... trust (or symmetrically, distrust) is a particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both before he can monitor such action ... and in a context in which it affects his own action."

A key aspect of Gambetta's definition is that trust is a probability of positive belief. It gives an indication of the expected outcome for future events [115]. From a political-science perspective, Fukuyama describes one of the most important functions of trust as being to facilitate honest and cooperative behaviour [124]. It is often easier for two mutually trusting parties to engage in an exchange than two mutually distrusting parties. For example, in a business setting, if a seller does not trust a buyer for honest and prompt payment, a transaction will simply not happen. Luhmann [127]

described this particular function of trust as a *complexity reduction tool* for societies, especially in the face of uncertainty and incomplete knowledge. Part of this social complexity comes from the presence of *risk*, which is a notion associated with uncertainty. He argued that trust is an essential means for handling risks and its existence enables us to face our daily life. Otherwise the risk of leaving the house and being hit by a car may be too great for one to even bother getting out of bed. Jøsang [114] shared a similar view and suggested that malicious behaviour is the primary reason for needing trust. In his model, a *passionate entity* may either be benevolent or malicious at its free will. In dealing with an unfamiliar passionate entity, trust serves as a prediction for the expected behaviour of the entity.

A recent study by Misztal [128] presented a comprehensive account of trust from a social perspective. Her main thesis is that trust is the key to maintain three types of social order: stability, cohesion and collaboration. She identified a form of trust that enforces each type of social order as an abstract concept, and also discussed practices that realize each form of trust. Trust that reinforces stability of a society is called *habitus*. This form of trust is associated with three common practices, namely, *habit*, *reputation* and *memory*. Habits include routine behaviour towards other people, taken-for-granted background assumptions in daily life, and rules of etiquette or rituals. All these types of habit repeat and relate past actions to the present, and therefore increase the predictability of social order. Reputation, also referred to as *social capital*, is a mechanism to assist a person in determining the *trustworthiness* of another. It helps reduce the social complexity by categorizing people into trustworthy and untrustworthy. Memory is similar to habit in that it allows past experiences to relate to the present. However, it involves the process of recollection, organization and recall of the past, and because it is simply a belief, it can easily be destroyed by new experiences. All these three practices improve social stability by enhancing its predictability, reliability and legibility.

Trust that promotes a cohesive society is in the form of *passion*. The basis of this form of trust lies in familiarity, bonds of friendship and common faith and values. There are three common sources of this trust: *family*, *friends* and *society*. Trust developed within a family is referred to as *basic trust*. It is upon this basis that a family provides a shelter against potential dangers in one's life – a fact learned by a person since being an infant. Friendship offers a different kind of trust, based on reciprocity and equality. It is developed through intimate self-disclosure and a feeling of shared solidarity, which are only found in close friendship, i.e. "real" friends. Trust provided by a society is based on networks of civic engagements and shared identity. It originates as the feeling of belonging together, commonly observed through religion, ethnicity or nationhood. In modern societies, the sense of belonging together ceases to be sufficient to establish societal trust, in addition active communication of autonomous members becomes a key to foster societal trust.

Policy is the third form of trust that improves collaborative order in societies. It serves as a means for members of a society to cope with the freedom of others. Misztal considered three issues relating to trust as policy: *solidarity*, *toleration* and *legitimacy*. Solidarity is based on the reliance on rational consensus in maintaining common interest in a society. It encourages people to participate, obey and cooperate. It sometimes can be achieved by rewards and sanctions, but the prime motivation is self-interest. However, differences among people exist in societies. To overcome the differences and enhance a cooperative order, toleration is an important ingredient in a society. It is the key to democracy, which respects diversity and resolves conflicts of interest through active communication. Tolerance therefore plays a vital role in achieving collaboration and cooperation in a society. Legitimacy is directly related to political trust, which can be loosely described as the "faith" people have in their government. This faith is obtained through the participation in political decision processes and continued monitoring of government performance. It creates a trustworthy, collaborative spirit between a state and its citizens.

Misztal's study reflects the complexity and broad reach of trust, even only in a social perspective. Disciplines like psychology, politics and economics have also been studying the phenomenon of trust to understand inter-personal, inter-organizational, and inter-national behaviour. Of course, not all these issues will be of direct relevance to modelling trust in computer science, but examining other disciplines does give us a more complete background to our applications of trust.

3.2.3 The basis of trust

Trust is an inherently dynamic measure. It can clearly be seen from both the security and sociological discussions above that there is no such thing as "permanent trust". Server A previously trusting server B as its public key authority may decide to cease the trust if B consistently vouches for bad public key bindings; a customer may start to distrust an online shop if the goods received do not meet their expected quality standards. The level of trust may increase or decrease depending on new knowledge and experiences learned from exercising the trust. A fundamental issue that must be addressed is how to "bootstrap" trust when there is no previous knowledge or track record available. In this circumstance, the only rational approach is to rely on external sources to provide information about the previously unknown party. Those external sources of trust may not only assist the establishment of initial trust, but may also affect the continued assessment of trust relationships. The common sources for initial trust are discussed below.

Recommendation, or "word-of-mouth" In real life, recommendation is perhaps the most commonly employed mechanism to assist decision-making in daily situations. It helps one infer trust decisions in an unfamiliar context by providing evaluations from others. Recommendation is typically obtained through friends and family, and sometimes through the media, institutions, or government. The trustworthiness of a recommendation depends heavily on its source, the source's authority in the context, and the source's responsibility and liability regarding the recommendation. Note that recommendation can also be negative. Recommendation is suitable for initiating a trust relationship. Nevertheless, it may be both unreliable and subjective [119, 128].

Reputation Reputation is another popular mechanism that people employ to deal with unfamiliar parties. Similar to recommendation, it does not require any prior experience with the party for reputation to be used to infer trustworthiness. It is thus suitable for establishing initial trust. Unlike recommendation, reputation is a collective opinion from the public regarding the untrusted party. Because of this nature, reputation is generally more reliable than personal recommendations. It is however subject to stereotyping and collusion, and can be deliberately manipulated to project a false image [128, 129, 121].

Experience Trust is intimately related with past experience. The basic assumption is that past experience provide a good indication of the outcome of future interactions. Past experience may be contributed from abstract, vague memory, or concrete, written records such as a transaction history or credit rating. The key issue is that it must provide a sensible relation from the past to the present. Experiences update one's degree of trust in another principal. Depending on the knowledge learned from previous interactions with the principal, the degree of trust may either increase or decrease. Experiences can also be provided by some trusted party, and such information may be as useful as recommendations. Note that reputation can sometimes be regarded as a form of collective experience if a principal builds its reputation primarily by interacting with others.

Miscellaneous There are numerous other mechanisms that affect a trust relationship and the trustworthiness of a principal. For example, cultural stereotypes may pose an inherent limit on one's trustworthiness [122]. In business, branding is an effective process for generating trust. It reflects the integrity and performance of a company through a concise representation, a logo, which easily reinforces people's memory about the company [115]. The behaviour of a principal or practices of a business may also have significant influences on people's trust. For example, if a company clarifies its responsibilities and provides a clear dispute resolution scheme, trust with its customers may be formed more easily [115, 130].

3.3 Conveying trust

It can be concluded from the previous discussion that trust is a complex concept. Fidelis does not attempt to define a unified trust model to satisfy all applications. On the contrary, it is believed that the diversity of applications needing trust makes it impossible to agree on a single unified view. A security application may require strong absolute trust, while "fuzzy" trust may be preferred in e-commerce applications which may be backed by dispute resolution and compensation plans so that business between complete strangers can be carried out. Based on this premise, Fidelis advocates a different approach, centering on the notion of *trust conveyance*.

3.3.1 Basic concept

Trust in Fidelis is defined as a set of assertions that a principal held with regard to another principal. An assertion may either be positive or negative, and in the latter case, we specifically call it *distrust*. Note that distrust is different from the absence of trust, which merely indicates lack of knowledge. Depending on the interpretation, an assertion may be treated as a principal's *belief* about other principals, or a weaker interpretation may simply treat an assertion as one's *statement* about others. An assertion is often associated with a specific *context*, where a context is defined as the situational conditions under which an assertion is expected to be interpreted with its intended meaning. From the perspective of the framework, there is no specific format for assertions. But as will be described later in Section 3.5, one approach is to represent them using first-order predicates, in the form of *named attributes*.

Fidelis trust is embodied in *trust statements*. A *trust statement* is a signed credential with a *truster* and a *subject*. The truster is the issuer of the trust statement; the subject is the principal the trust statement concerns. A trust statement represents a trust relationship between the truster and the subject, and is signed by the truster. The signature is a crucial component in a trust statement, which serves two purposes. First, it proves the authenticity of a trust statement; second, and more importantly, it indicates the explicit source of a trust statement. A signature, both digital or non-digital, creates a binding relationship between the signer and the signed entity. The basic intent of a signature is to prove the consent of the signer with respect to the signed entity. Since a signature is assumed to be unforgeable which only its owner can produce, a signed trust statement identifies its truster. Signatures additionally have the property of non-repudiation [131]. Recall that from the discussion of recommendation in Section 3.2.3, claiming responsibility and liability increases the trustworthiness of a recommendation. Likewise, the trustworthiness increases if the signature of a trust statement offers a non-repudiation guarantee.

Trust is said to be *conveyed* if one principal passes a trust statement to another. Such an instance is called a *conveyance instance*. A *conveyance source* (or *source*) is defined as the principal who transfers a trust statement in a conveyance instance, and a *conveyance target* (or *target*) is defined as the principal who receives the trust statement in a conveyance instance. A source may or may not be the truster of the conveyed trust statement, although it is often the case that the truster acts as the source for its own trust statements. Similarly, the target need not be the subject of the trust statement it is receiving. Figure 3.2 illustrates an instance of conveyance, in the context of public key certification. A public key certificate can be considered as a trust statement: the truster, in this case Bob, certifies the key of the subject, Alice. Another principal, Cindy, may somehow learn this assertion and decide to propagate it as a conveyance source, to David, the target. The world of principals forms a *conveyance network*, where principals transfer, exchange, and receive trust statements from one another.

The trust conveyance approach builds on three basic principles:

- *Trust is subjective.* Every principal has the discretionary power to make its own trust decisions, which may be based on the trust statements it believes.
- *Trust is specific.* Every trust statement has a specific context that defines its scope of use. It is however up to the conveyance target of a trust statement to interpret its context.

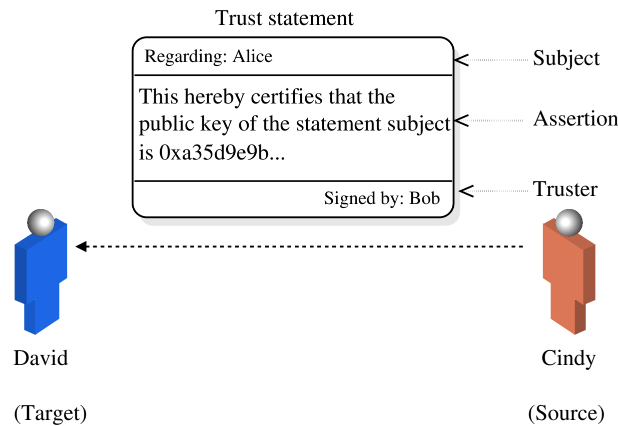


Figure 3.2: Conveying trust

- *Trust is dynamic.* Trust statements should be subject to some validity conditions so that ones representing outdated knowledge will be invalidated.

Besides these three principles, Fidelis imposes no further assumptions on the concept of trust. In particular, it does not force a single-minded view of trust. Instead, every principal has complete freedom to choose its trust model, which may have a definition of trust level and/or methods for computing trust. In this regard, trust statements serve as an interface to communicate with other principals. This departs sharply from other approaches which attempt to define domain-specific trust models. A brief discussion will be given later comparing the trust conveyance approach and other approaches.

3.3.2 Validity

As discussed previously, trust is a dynamic concept, evolving with experiences and updated knowledge from peers. A trust statement is a concrete representation of the contextual trust, and therefore must be subject to the evolution of the trust it represents. To address this, a *validity condition* is included for every trust statement. The idea is that this mechanism reserves the rights for a truster to invalidate its trust statements where necessary, and a truster may decide to issue new trust statements upon invalidation.

There are a number of techniques for expressing validity conditions. X.509 [6] specifies a coarsely grained validity period for its certificates, with the assumption of synchronized clocks at the global scale. It uses a revocation list mechanism to invalidate a certificate prior to the end of its validity period. Micali [132] describes techniques for improving the computation and communication cost of revocation based on Merkle trees. Other work on applying tree structures to improve revocation include [133, 134, 135, 136]. OASIS [15, 67] uses efficient asynchronous messaging to maintain real-time validity of its certificates. This is complemented by the infrastructure support for network failure detection.

The conveyance model does not prescribe a particular validity mechanism. Different validity mechanisms deliver different degrees of guarantee, and it is an application issue to determine the validity strength of its trust statements. The model however requires a validity method to follow a *determinism principle*. The principle is that the validity of a trust statement cannot be negated once it is *guaranteed*. A consequence is that the processing behaviour will be deterministic, with no "sudden surprises". These semantics are desirable especially in a widely distributed system where network failure and partition are inevitable.

As an example, a possible validity mechanism that exhibits deterministic behaviour would be a

simple validity period without revocation lists. The absence of revocation lists ensures that a trust statement only invalidates at the end of its period, thus the validity guarantee cannot be broken by any means. This is an example of an *offline* mechanism, where the validity of a trust statement is maintained independently of the availability of the network. A family of online mechanisms is supported in the Fidelis Policy Language, and will be discussed in Section 3.5.4.

3.3.3 Discussion

There have been several attempts to model trust in the past. Abdul et al. [119, 120, 123] proposed trust models for general distributed systems, for virtual communities and for information retrieval needs. Their models compute trust values based on the degree of trust of recommenders. Some of their models include protocols for updating experiences and recalculating trust values. Jøsang [108, 137, 138] attempts to capture trust using *subjective logic*, that computes an *opinion value* along three axes, belief, disbelief and uncertainty. He describes a scheme for combining opinion values and a protocol for initiating a trust relationship and evaluating trust values. Manchala [139, 116] presents a trust model for e-commerce, which computes trust values to include parameters such as transaction cost, transaction history, customer loyalty, etc. His model incorporates the concept of risk analysis and is based on a fuzzy logic for inferring trustworthiness. Similar to others, he also described a protocol for maintaining trust values. Marsh [140], in his PhD thesis, describes a comprehensive modelling of trust with a focus on the sociological properties. In his model, he attempts to capture many facets of trust under one formalism, such as risk, confidence, expectancy, cooperation, etc. The underlying idea is similar to that of others' derivation of trust values.

There exist many other similar attempts for different application areas [141, 142]. It is unlikely that a unified model will ever exist to satisfy individual needs. Instead of proposing yet another trust model, the trust conveyance model attempts to provide a framework in which these trust models may interoperate and cooperate. One of the primary reasons for defining trust models is to create a basis for participants to infer trust-related decisions. In large distributed systems, there are three difficulties with this approach. First, as discussed previously, the notion of *trust* differs significantly depending on the nature of applications. Second, such models typically require some monitoring mechanism to ensure every participant's compliance. Distributed monitoring is however subject to operational availability of the infrastructure and general scalability problems. Third, autonomous participants may have different trust assessment schemes, which include subjective opinions and errors. It is unclear how a trust model can be enforced in the light of principal autonomy.

The conveyance model focuses on the secure propagation of trust statements between principals. As in human society, this mechanism is often taken for granted, e.g. from friends, the media, background, instinct, etc. The conveyance model formalizes such a mechanism for distributed environments. This allows every principal to define its trust model, and to interoperate with others through common agreement on the models. Thus, the trust conveyance model effectively complements rather than replaces those trust models to enable their applications in distributed environments.

3.4 Identity

Fidelis adopts a key-centric approach which identifies principals by public keys. A *principal* may represent a person, an organization, or a computer process, etc. The key-centric treatment does not distinguish the actual entity represented by the principal, but instead insists that a principal must *control* (i.e. speak for) a public key pair. Every principal may freely generate a public key pair at any time. The generated public key can then be used as an identifier for the principal. Global uniqueness is guaranteed by the fundamental requirement of the chosen public key cryptosystem, which ensures no collision of keys is possible, given a sufficiently large entropy, e.g. 1024 bits. A prerequisite assumption for this key-centric approach is that every principal should exercise good safeguarding practice for its private keys, which is a typical assumption for public key cryptography. There are measures to encourage and enforce this prerequisite requirement. These will nevertheless not be discussed here.

A principal may control multiple keys simultaneously. It is a common practice to limit the damage of a possible compromise of a key by constraining its use. Suppose a principal has a key pair for e-mail communication and another for workstation login. If the former key pair is compromised, it would only affect its e-mail usage but cause no damage to workstation access. The same physical principal is therefore allowed to be identified by multiple public keys. Each public key is treated as a separate instance of the principal.

This key-centric approach provides a possibility for anonymity. Provided a principal generates a fresh public key on every anonymous access and, by requirement, there is no mathematical relationship between any two keys, the principal can effectively "hide" its identity using a new public key. It is important to note that this mechanism alone is not sufficient to prevent analysis based on linked access patterns and attacks based on collusion. Public keys as principal identifiers merely provide a ready source of pseudonyms. For a more detailed discussion on these privacy issues, please refer to Section 7.4.

3.4.1 Discussion

The trust conveyance model places two requirements on naming support. First, a conveyance target must be able to validate the authenticity of a trust statement based on the identity of the truster. Second, every principal must be uniquely identified in the system. Failure of this introduces ambiguity and prevents communication between arbitrary pairs of principals.

A possible approach to satisfy these requirements is to deploy a global naming system and couple it with a public key infrastructure. The original plan of the X.500 directory service is a prime example of this approach. The idea is to associate every principal with a hierarchical name. Association between a public key and a name is then certified by some Certification Authority (CA). There are several problems with this approach as discussed in the literature [18, 143]. Hierarchical namespaces are introduced to address the scalability problems associated with flat namespaces. However, this in itself requires a standard hierarchy so that names can be meaningful to every principal. This is often difficult, if not infeasible, since every community will have a preferred naming structure, for intuition and convenience reasons. The partitioning of namespaces must be permanent to ensure the validity of a name. Evolution of a namespace will invalidate all of its dependent namespaces in the hierarchy. Furthermore, hierarchical namespaces require naming authorities at each level to ensure unique allocation of names. This centralized management, even scoped locally, may eventually become a problem in large-scale systems with potentially thousands of users.

A more significant problem is global key management. Because of the hierarchical nature, trusting a key binding implies trusting all the intermediary authorities along the chain to the root authority. Breach of security at an authority will therefore have a propagating effect to all its descendants. The root of the hierarchy becomes an attractive point for attack, since breaking the root will enable an adversary to control the entire structure. This problem is largely due to the implicit assumption in X.509 where a naming hierarchy is assumed to reflect the trust hierarchy for key certification. This aggregates trust towards the root of the hierarchy, i.e. the higher up in the hierarchy, the stronger trust assumption is required. This rigid assumption precludes the dynamic nature of today's distributed applications, where trust relationships tend to be complex and constantly changing. More importantly, it forces applications to adopt a single trust structure.

The key-centric approach pioneered by modern key-oriented access control schemes presents an elegant solution. Public keys are mathematically designed to be globally unique by nature. The probability of two keys clashing is negligible. Additionally, public keys do not need to be kept secret. These two properties meet the basic requirements of identifiers. However, the real value of the key-centric approach is the avoidance of names. An important observation is that names are mostly for the convenience of humans [18, 144, 143]. People are used to identifying others by names – a practice learned from the early days of one's life. While natural for humans, names are of little value for computer systems. In an open system, the strongest guarantee is the knowledge that the remote communicating party controls a particular private key. Proving the name is a secondary action which requires a secure binding from the key to the name. The key-centric approach does not deal with names

and hence eliminates the need for name management. A desirable consequence is the independence from central trusted third parties to certify the authenticity of keys. If a principal can be identified, its key will be known. This fits naturally with the trust conveyance model, where a public key in a trust statement can both identify its truster and verify its integrity.

Although the key-centric approach solves the global naming problem, on the other hand, it introduces another problem due to its source of principal identifiers. Since by assumption, every principal may generate a fresh key pair and use the public key as its identifier, the public key is inherently anonymous. For example, if a principal is blacklisted for financial fraud, he/she may simply generate a new key pair, essentially creating a new identity, to avoid being caught.

In Fidelis, this problem is considered a policy issue. It is up to each individual service to decide whether anonymous public keys are accepted. If a service requires persistent names, it may demand a principal to present trust instances issued by some trusted authority, e.g. the Government registrar providing a name-certification service, linking public key identifiers to names. To bind a name to a public key in an authoritative manner, the authority should typically follow rigourous procedures, identifying both the ownership of the key and the name, and possibly some additional attributes that are asserted.

3.5 The Fidelis Policy Language

The Fidelis Policy Language is a language designed to facilitate the trust conveyance model. It is intended for use by principals in a conveyance network to specify their policies regarding trust statements. There are two kinds of policy in Fidelis: a *trust policy* defines the relationships between trust statements; an *action policy* relates an action with trust statements. This section describes the syntax of the language and provides an informal semantics. Note that the use of this language is not compulsory; a principal may hard-code policies, or use other languages according to their resources and need. The language serves as a general reference for common applications to adopt the trust framework.

3.5.1 Principals

There are three types of principal. A plain principal is specified as its public key. A *principal group* is specified as a set of public keys. A *threshold principal* is specified as a set of public keys, with a threshold value of the minimum number of representative principals in the set. The syntax for specifying a principal is:

$$\begin{array}{lcl} \textit{principal} & ::= & \textit{public key} \\ & | & \textbf{\{}\ \langle\textit{principal set}\rangle\ \textbf{\}} \\ & | & \textit{integer}\textbf{-of}\ \textbf{\{}\ \langle\textit{principal set}\rangle\ \textbf{\}} \\ & | & \textbf{self} \\ & | & \textbf{any-key} \end{array} \tag{3.1}$$

The literal representation for a public key is a hexadecimal string. This assumes some encoding scheme is employed to produce a hexadecimal value for either the public key in full length, or a hash of the key. The actual encoding scheme (e.g. Base64) and/or hash algorithm (e.g. MD5) used to produce the string representation of a public key are considered as implementation details. It is left to the choice of the implementor.

Group principals are conjunctions of principals. The intuition is to treat the principals in a group as a single, logical principal. This enables representation of concepts such as joint statements. A trust statement signed by a group principal is semantically identical to the same trust statement individually signed by all members of the group and aggregated together.

A threshold principal is a special type of group principal. While a plain group principal represents the entire set of group members, a threshold principal represents a subset of a group, with a minimum

number of principals in the set. The minimum number is the threshold value, specified as an integer. The threshold construct enables the specification of threshold schemes. A common commercial threshold scheme would be that a company cheque typically requires two or more signatures for it to be valid. An example threshold principal is:

```
2-of {0x023296de..., 0xca91f513..., 0xf6994a9b..., }
```

The principal set for group or threshold principals may be specified literally, as shown above, or refer to a variable which will be bound during evaluation. This is useful for large groups or dynamic groups backed by databases. Its syntax is,

principal set	::=	*public key*, ...	(3.2)
	|	*variable*	

The language provides the **self** keyword for representing the public key of the policy owner. In theory, there is no difference between a policy owner and the rest of the world – a literal representation can be used to identify the policy owner. It is however sometimes useful to late-bind the policy owner at deployment rather than at specification time. This allows some degree of centralized policy management, whereby an authority may define a standard trust policy and distribute it to participating principals for enforcement.

An **any-key** keyword is provided as a wildcard for public keys. It is intended for policies that need not consider specific trusters or subjects. For example, a policy may state *any person certified by the local authentication server may log onto a workstation*.

3.5.2 Actions

An action encapsulates computation that may be subject to policies. As a motivating example, consider an access control scenario, where an access control monitor in an operating system needs to determine if a requester is allowed to read a file. An intuitive action representation would be `read_file`, whose context includes a file name and a requester. Actions may also be high-level and abstract. For example, an online shop may represent the execution of a transaction which consists of a number of low-level read and write operations as a single action `do_transaction`.

The notion of actions is typically defined differently across applications. As briefly shown in the previous paragraph, an action may directly correspond to a method invocation, or it may be a general trust query. To satisfy these diverse needs, the Fidelis language generalizes actions as parameterized predicates. The syntax is:

action spec	::=	⟨*action name*⟩ **(** ⟨*parameter spec*⟩**,** ... **)**	(3.3)
parameter spec	::=	⟨*type*⟩ ⟨*name*⟩	(3.4)
action instance	::=	⟨*action name*⟩ **(** ⟨*parameter instance*⟩**,** ... **)**	(3.5)
parameter instance	::=	*value*	(3.6)

An *action specification* (3.3) consists of a name and an optional list of formal parameters. The name is given as a string, and a formal parameter consists of a type specifier and a name (3.4). The name of a formal parameter is scoped within the action specification and must be unique within the scope. There is no built-in type system in the language. It is deemed to be an implementation and deployment issue. There are numerous choices in programming languages (e.g. Java, C, C++), database management systems (e.g. SQL, OQL/ODL [145]), and distributed middleware (e.g. CORBA, DCOM [146]). The type system used in a policy must be identified when it is processed. For descriptive convenience in this chapter, a simple type system consisting of only primitive types, including `int`, `float`, and `string`, will be used. Public keys will have a special primitive type `pubkey`.

An *action instance* (3.5) is an instance of an action specification. It is defined by a name and a list of parameter instances. The name refers to an action specification, and the parameter instances must match the specification. A parameter instance is given as a literal value in the value space of the parameter type.

3.5.3 Trust specification

Recall from Section 3.3 that a *trust statement* carries assertions about a subject held by a truster. The Fidelis language employs a similar abstraction for expressing assertions as for actions. Assertions are represented in the form of parameterized predicates. The syntax for trust specification is thus similar to action specification (3.3):

statement spec	::=	⟨*statement name*⟩ **(** ⟨*parameter spec*⟩**,** ... **)**	(3.7)
statement name	::=	*string*	(3.8)
trust spec	::=	⟨*statement spec*⟩	(3.9)

A *trust specification* (3.9) is defined as a *statement specification* (3.7), which is in turn defined by a locally scoped name and a list of formal parameters. As with actions, the parameter list is optional. Some assertions are simple and narrowly scoped, and can be expressed without parameters. An example would be `paid()` in an online purchase session. A customer who has paid for a purchase may be certified by the accounts department of the selling company, and its delivery department, based on this assertion, may then arrange for purchase dispatch. Such trusts are Boolean, i.e. only "believed" or "not believed".

A more reusable trust specification involves parameters. An example in identity-based access control would be:

```
user (string user_id)
```

which represents the belief that a subject is recognized as the user `user_id` by a truster. This assertion could be, for example, signed by an authentication server and passed to the point of access as an access token.

It is important at this point to distinguish *trust statement instances* from *trust specifications*, which were collectively referred to as trust statements previously. A trust specification is not bound to a specific truster and subject. Only beliefs are specified. A principal instantiates a trust specification in the capacity of a truster, regarding its belief concerning another principal. A concrete trust statement is referred to as a *trust statement instance* or simply *trust instance*. In the Fidelis policy language, the syntax component for referencing trust instances is given a name *trust use*. Trust uses are designed for matching trust instances in a policy, and have the following syntax:

trust use	::=	⟨*statement use*⟩ **:** ⟨*truster*⟩ → ⟨*subject*⟩	(3.10)
statement use	::=	⟨*statement name*⟩ **(** ⟨*placeholder*⟩**,** ... **)**	(3.11)
	|	**any-statement** [**as** *variable*]	
placeholder	::=	*variable*	(3.12)

A *trust use* (3.10) is defined as a *statement use*, associated with a truster and a subject. A statement use (3.11) references a trust specification by a name and has a list of parameter placeholders. A parameter placeholder (3.12) is a variable whose value is provided by the actual parameter in a trust instance at evaluation. Parameters are for matching and extracting values across trust instances in a trust policy. This will be described further in Section 3.5.5. The truster and subject of a trust use have the following syntax:

$$\begin{array}{rcll} \textit{truster} & ::= & \langle \textit{principal specifier} \rangle & (3.13) \\ \textit{subject} & ::= & \langle \textit{principal specifier} \rangle & (3.14) \\ \textit{principal specifier} & ::= & \langle \textit{principal} \rangle \ [\ \textbf{as}\ \textit{variable}\ ] & (3.15) \\ & | & \textit{variable} & \end{array}$$

They are defined as *principal specifiers*, where a principal specifier may either be given as a principal syntax item (3.1) or a variable. For example, a trust use

```
user(user_id) : 0xb3d981235 -> any-key
```

would match a `user` trust instance signed by principal 0xb3d981235 for any principal. Note that the keyword **any-key** matches any principal, see Section 3.5.1.

A truster and subject could also be given as placeholder variables. This enables the matching of principals across trust instances and allows additional conditions on these principals. See Section 3.5.5 for examples. A principal specifier may also be associated with a variable, in which case the value for the variable will be bound to the actual matching principal at evaluation, for example, the actual set of principals that forms a satisfied threshold principal.

A trust use that matches any trust instance may be specified using the keyword **any-statement**. This construct is fairly infrequently used in practice as over-generalization generally reduces its applicability. A possible use is to specify *blind delegation*, i.e. relaying whatever a truster asserts. Section 3.5.5 includes an example of **any-statement** to construct blind delegation policies. It is also possible to refer to the particular trust statement instance matched by an **any-statement** using the placeholder mechanism. For example,

```
any-statement as t: 0xb3d981235 -> any-key
```

allows variable `t` to refer to the actual trust instance signed by 0xb3d981235 for any subject.

3.5.4 Validity conditions

Every trust statement instance has a validity condition as discussed in Section 3.3.2. Recall that the fundamental requirement for a validity condition is to exhibit deterministic behaviour, i.e. there cannot be exception clauses causing a guaranteed validity of a trust statement to negate. The language supports one offline and three online validity methods. The syntax for validity conditions is:

$$\begin{array}{rcll} \textit{validity} & ::= & \langle \textit{offline validity} \rangle & (3.16) \\ & | & \langle \textit{online validity} \rangle & \\ & | & \textbf{always} & \end{array}$$

Besides offline and online validity methods, a keyword **always** is provided to express permanent, absolute belief, e.g. family relationships. It is however rarely used as absolute, constant trust is rare.

The offline method specifies a *validity period*. Its syntax is shown below:

$$\textit{offline validity} \ ::= \ \textbf{from}\ \langle \textit{time spec} \rangle\ \textbf{to}\ \langle \textit{time spec} \rangle \qquad (3.17)$$

The semantics for a validity period is that a trust instance is *guaranteed* to be valid for the specified duration. This means there exists no mechanisms to invalidate the trust instance during this period. The trust instance is considered to be invalid once the validity period is over. This semantics is similar to that in SPKI [18], and is dramatically different from X.509 [6], where the validity period only serves as a "hint" for the validity of a certificate since it may still be revoked by a certificate revocation list (CRL). The *time specification* denotes a time instant, specified as a constrained ISO 8601 format [147]:

CCYY-MM-DD hh:mm:ss

where *CC*, *YY*, *MM*, *DD* represent the century, year, month, and day, and *hh*, *mm*, *ss* represent the hour, minute, and second respectively. A right-truncated format could be used if the time components are of interest, e.g. 2002-04-15.

The three supported online validity methods are: timed credential revocation list, timed renewal and status check. Their syntax is:

online validity	::=	**CRL** [**async**] **at** ⟨*address*⟩**,** ...	(3.18)
	|	**renewal** [**async**] **at** ⟨*address*⟩**,** ...	
	|	**status at** ⟨*address*⟩**,** ...	

The format for *address* is implementation-dependent. Possible choices include, IP, DNS name, and URL, etc. Fidelis allows online validity information to be either fetched from designated locations or delivered asynchronously to the verifiers. In the latter case, a verifier needs to register for notification at the specified addresses and exposes an interface for receiving asynchronous messages.

A timed credential revocation list (t-CRL) contains a list of references to revoked trust instances, with an expiry period. A reference may be in the form of a cryptographic hash of a trust instance, or may be a reference number linked to a trust instance. The expiry period represents the temporal lower- and upper-bound for which a t-CRL can be used. When a trust instance whose validity is determined by t-CRLs is processed, it is required that a valid t-CRL covering the present time must be available, and the expiry period of a new t-CRL must not overlap with the current t-CRL. The former requirement prevents processing of a trust instance if its t-CRL is unavailable. The latter requirement prevents a sudden "change of mind" by the truster. Therefore, if a trust instance (or its reference) is not contained in the current t-CRL, it is guaranteed to be valid at least until the end of the expiry time of the t-CRL. A t-CRL can be facilitated through asynchronous messaging, in which case, the delivery of a new t-CRL can take place any time before the current t-CRL expires.

A timed renewal is similar to the offline method, but with an automatic expiry extension at the end of the validity period. Two conditions similar to the t-CRL processing exist for timed renewal processing to ensure determinism. First, a valid renewal must be present when processing a renewal-based trust instance. Second, the extended and the current period must not overlap, but they need not adjoin. This leaves no possibility to negate previous validity conditions but, since validity periods are not required to join, it does permit an "uncovered" period, i.e. time when a trust instance is "temporarily unavailable". Like t-CRL, timed renewal can also be implemented using asynchronous messaging. The delivery of a status notification can occur any time before the current renewal period expires.

The third online method is a status check. This represents a status query at some designated address. The query consists of the reference to a trust instance (e.g. a serial number) and the result is a Boolean status value and a timestamp. The returned status is expected to have the lifetime of a single policy computation. Another status check needs to be performed if the same trust instance is processed again. If a query is not possible, e.g. due to network failures, the trust instance will not be processable. This semantics enforces a deterministic behaviour within a single computation, i.e. if the status of a trust instance is determined at one point during a policy computation, it will remain so until the computation finishes. It is important to note that asynchronous messaging is not applicable for a status check because prevention of message delivery would be a sufficient attack, unless some heartbeat mechanism is employed.

The language does not attempt to offer a complete list of validity options, rather, these methods exist to promote interoperability. An implementation may however support other specialized validity methods where these standard methods are not appropriate.

3.5.5 Trust relationships

The most important aspect of the language is the specification construct for *trust policies.* Generally speaking, a trust policy defines a principal's belief about another principal, i.e. a *trust relationship.*

The trust policy construct offers building blocks for capturing common factors of trust, including recommendation, reputation and to a certain extent, experiences, as previously discussed in Section 3.2.3.

A trust policy may serve two purposes. First, it defines conditions for trust establishment. For example, Alice may specify conditions that must be met before she trusts Bob to sell books. Bob may approach Alice to obtain her trust by presenting "proofs". If Alice's conditions are satisfied, she establishes a trust relationship with Bob by creating and signing a trust instance. Second, it assists trust decision-making. Continuing the previous example, suppose Cindy wishes to determine Bob's trustworthiness for selling books. She may approach Alice with some beliefs she holds about Bob. Alice may then reply to Cindy if she thinks Bob is trustworthy according to her own policies.

Before the syntax for trust policies can be described, we shall first define *trust templates*. A trust template serves as a template for creating new trust instances, specifying values to be bound to parameters upon instantiation. It has the following syntax:

trust template	::=	⟨*statement name*⟩ **(** ⟨*parameter*⟩**,** ... **) :**	
		⟨*truster*⟩ → ⟨*subject*⟩	(3.19)
parameter	::=	⟨*parameter instance*⟩	(3.20)
	|	*variable*	

A trust template is essentially a partially instantiated trust instance. It has a name, a list of *parameters*, and a pair of truster and subject. Each parameter (3.20) is defined as either a parameter instance (3.6) or a variable. Recall that a parameter instance is a concrete value of the type of the parameter. Truster and subject are as defined in (3.13) and (3.14) respectively.

The basic structure of a trust policy consists of a set of trust uses (3.10) matching the set of prerequisite trust instances for the new trust instance. The policy may optionally include another set of trust uses for matching trust instances whose existence prevents the creation of the new trust instance. Conditions and rules may be specified to constrain parameters in trust instances and to set values for variables. Additionally, it is possible to associate specific actions and/or validity conditions with new trust instances. The syntax of a trust policy is:

trust policy	::=	[⟨*trust use*⟩**,** ...] [**without** ⟨*trust use*⟩**,** ...]	(3.21)
		asserts ⟨*trust template*⟩	
		[**where** ⟨*conditions*⟩] [**set** ⟨*assignments*⟩]	
		[**grants** ⟨*action template*⟩ **,** ...] [**valid** ⟨*validity*⟩]	

where *validity* is defined in (3.16), *conditions* and *assignments* are described later in Section 3.5.7. As an example, a simple trust policy may be specified as,

```
T1(a, b): self -> Z, T2(b, c): Y -> Z  asserts T3(c): self -> Z
```

where trust names are `T1`, `T2`, etc; principals are given in uppercase letters instead of literal public keys for readability; variables are in lowercase letters. This policy states that the policy owner (namely, `self`) believes T3 regarding principal Z, provided she believes T1 about Z, and Y believes T2 about Z at the same time. The language features a *variable matching* rule, whereby the value of all occurrences of the same variable must match. Therefore to obtain a T3 instance according to the above policy, valid instances of T1 and T2 with matching parameter instances must be presented. For example, assuming the policy owner is X, it would be sufficient to present

```
T1(1234, "pay"): X -> Z
T2("pay", "alice"): Y -> Z
```

and the new trust instance will be:

```
T3("alice"): X -> Z
```

Presenting the following trust instances will however fail because of mismatched parameters:

```
T1(1234, "pay"): X -> Z
T2("buy", "alice"): Y -> Z
```

Trust is a non-monotonic concept [101, 137, 116], e.g. an entity can be believed to be malicious. Recall from Section 3.3.1 that the framework has the notion of *distrust*. The **without** clause is the mechanism in the language to support this notion. It allows negative comments/recommendations to be considered. Effectively, it means that the trust instances matched by the trust uses in the **without** clause must not exist for the trust policy to be evaluated with a positive result, i.e. certain negative trust instances must not exist. A typical use is to implement a "blacklist" mechanism to prevent distrusted principals causing further damage to others. A real-life example is the Better Business Bureau, which in addition to listing good businesses also often lists bad businesses as a warning for consumers.

The variable matching rule provides a coarse-grained constraining instrument for parameters in trust instances. Fine-grained constraints can be specified through the *conditional expression* in a **where** clause. A conditional expression operates on: (1) parameter variables in trust and distruct uses, and (2) environmental variables. An *environmental variable* is a typed name-value pair, whose value is supplied externally at evaluation. An *environment* consists of a list of environmental variables. The syntax for conditional expressions is specific and may be local to every policy specification. The only requirement is that a conditional expression must be *side-effect free*. Expressions used in this thesis include operators for: arithmetic, comparison, logical connectives, regular expressions, groups and principals. In particular, it allows embedded SQL statements, enclosed in a pair of double-square brackets (`[[ ]]`). Variables may be used directly in an embedded SQL statement, provided they are preceded by a `$` character. Section 3.5.7 describes the syntax in more detail. Some example expressions are:

Comparison operator: `a == 1234`
Regular expression: `b =~ "/etc/.*"`
Logical operator: `a == 1234 && b =~ "/etc/.*"`
Principal operator: `c in {0xca04156f, 0x15ba430d, 0x528ba0bf}`
Embedded SQL: `[[ SELECT * FROM users WHERE user_id = ‘wtmy2’]]`

A policy evaluation may result in a new trust instance. A parameter in the result trust instance can be given directly in its trust template. If the value of a parameter depends on the context of policy evaluation it can be set either through the variable matching mechanism or explicitly in an *assignment expression*. An example is:

```
T4(cust): self -> Y
asserts T5(4000, limit): self -> Y
set limit = [[SELECT limit FROM credit_limits WHERE cust_id = ‘$cust’]]
```

The first parameter of a `T5` instance is set with a predefined value, '4000', while the second parameter is set from the result of an embedded SQL query. Like conditional expressions, the syntax for assignment expressions is also application-specific. Use of proprietary languages does not impede the interoperability since the evaluation is entirely internal to the principal.

The interface between the policy language and the conditional/assignment expressions is through *variable bindings*. The evaluation of a conditional expression, which is side-effect free, is guaranteed to yield a deterministic output. Since an assignment expression may create or modify variable bindings, it is required to be evaluated *after* the conditional expression. This ensures a well-defined behaviour for the evaluation of both conditional and assignment expressions.

By default, the processing semantics ensures the validity condition for a new trust instance is the *weakest validity condition* among those prerequisite trust instances. The rationale is that if a prerequisite trust instance becomes invalid, the dependent trust instances should also become invalid.

The rules for deriving the weakest validity condition are described in Section 3.5.8. As a motivation, suppose the instances of T1 and T2 in the previous example have validity conditions:

Trust statement instance	Validity condition
`T1(1234, "pay"): X -> Z`	from 01/04/02 to 01/04/03
`T2("pay", "alice"): Y -> Z`	from 10/05/02 to 20/05/02

The new T3 instance (namely, `T3("alice"): X -> Z`) will then have a validity from 10/05/02 to 20/05/02. The validity condition can also be explicitly specified as part of a trust policy, using the **valid** keyword. In this case, the specified validity condition will override the default semantics. Effectively it implies the validity of a new trust instance is independent of the validity of the prerequisite trust instances. This is useful to express belief about historical events, e.g. *order ID 2504 has been processed*.

The rest of this section presents some examples to demonstrate trust policy specification. One example motivates and describes the use of the **grants** keyword, which has not been covered so far.

Example: Bootstraping trust Bootstraping trust is also known as an axiom or basic belief. It's a fundamental belief held by a principal and is intended as a ground rule from which one's trust decisions are inferred. It often expresses a fact or an "instinctive" belief, i.e. a belief needing no questions, e.g. *"Joe Bloggs is Jon Bloggs' father"*. In the policy language, bootstrapping trust is expressed as a trust policy with no prerequisite trust uses. Consequently, it must have an explicit validity clause.

Consider an authentication service. Principals identify themselves as public keys. A bootstrapping trust statement may be testifying whether a principal is recognized as a local user, which may be specified as:

```
user(string user_id)
```

The service may then define an explicit mapping from a principal public key to a local user identifier, stored in a relational database. Assume that the database stores every account under a tuple (`username`, `key`), where `username` is a local user identifier and `key` is the public key of the user. The policy may then be specified:

```
asserts user(user_id): self -> p
set user_id = [[ SELECT username FROM user_db WHERE key = '$p' ]]
validity status at fidelis.cl.cam.ac.uk
```

When a principal invokes the service, the service first constructs an environment containing a binding for `p` – the requester's public key. It then consults this trust policy which performs a local database query to determine the corresponding user name. This processing results in a new trust instance proving the service's knowledge about the requester. This can then be used by other services for access control purposes. Note that the policy is written with an assumption that there is a unique user ID/public key binding. The assignment language is therefore expected to handle multiple results from the SQL query, e.g. fails if there is more than one result. This is however an implementation issue.

Example: Recommendation Consider a real-world example. The Hong Kong Jockey Club has a membership rule whereby a candidate member must be endorsed by two voting members. A voting member has the right to propose and second for membership and there are currently around 200 voting members in the club.

One approach assumes that the jockey club specifies trust statements for regular and voting members, i.e. *the subject is a regular member or voting member*, and another trust statement represents endorsements. The membership rule can then be expressed as a trust policy:

```
voting_member(): self -> p1, endorsement(): p1 -> p,
voting_member(): self -> p2, endorsement(): p2 -> p
asserts member(): self -> p
where p1 != p2
```

In plain English, the above policy states that if principal `p` is endorsed by a voting member `p1` and also by `p2`, and `p1` and `p2` are different, then `p` is accepted as a regular member. There may exist some other trust policies that define how a regular member may become a voting member, but this is outside the scope of this example.

Another approach makes use of threshold principals. It specifies trust statements for members and endorsements. In addition, it assumes the member information is stored in a relational table, `members`, with these fields:

Field	Type	Description
id	pubkey	Principal identifier
vote	Boolean	Has the voting right?

The membership rule can then be specified as follows,

```
endorsement(): 2-of {voters} -> p
asserts member(): self -> p
where voters < [[ SELECT id FROM members WHERE vote=TRUE ]]
```

This trust policy states that if principal `p` is endorsed by two members who have voting rights, `p` is then accepted as a regular member. The operator `<`, described in Section 3.5.7, determines whether the actual trusters of an `endorsement()` instance satisfy the threshold condition, given the group defined by the SQL query.

Comparing these two approaches, while the former captures the real policy, it is cumbersome and less straightforward. It also has a scalability problem; if the required number of voting members is higher, it will become less maintainable and more error-prone. The latter approach faithfully models the real policy, and has attractive maintainability characteristics. It nevertheless requires external database support.

Example: Authorization trust An authorization certificate in key-oriented access control can be considered as a special kind of trust instance, where a certificate holder is trusted with certain authorizations. In this regard, an authorization can be thought of as a refined form of trust [117]. The Fidelis language supports this type of trust policy through the use of the **grants** keyword, which allows a direct binding of action instances with a trust instance.

Consider a banking service. Suppose the service issues trust instances to every customer, asserting the ownership of their accounts. This is specified as `owner(ac)`, where `ac` gives an account number. It also issues special trust instances, `capabilities()` embodying the authorization, perhaps carried in a smart card. Assuming an account owner is allowed to query balance, withdraw and deposit money, a trust policy may be:

```
owner(account_no): bank -> p
asserts capabilities(): bank -> p
grants balance(account_no): p, withdraw(account_no): p, deposit(account_no): p,
      ...
```

A customer possessing an instance of `capabilities()` can present it to access points of banking services, e.g. cash machines. At each access point, it would only need to determine if the requested action is contained in a `capabilities()` instance.

Some advantages of this key-oriented style of access control have been described in Section 2.4. Briefly, first, it simplifies the access control monitors – essentially an access control monitor only needs

to verify the integrity of a trust instance and examine if the trust instance contains the authorization. This simplicity implies potential deployment in lightweight environments, such as on mobile devices. It also speeds up access control decisions since there is no complex policy to query. Furthermore, it features an appealing scalability characteristic as access control policies effectively are distributed to every principal in the form of trust instances. However, as will be discussed in Section 3.5.6, quite often it is not always appropriate to use such access control schemes.

Example: Delegation of trust Delegation in security often refers to the delegation of rights, which enables authorization propagation from a principal to another. In Fidelis, a different form of delegation can be expressed, known as the *delegation of trust*. This refers to the mechanism that a principal asserts beliefs it learns from others, passing them on as its own beliefs.

Consider an example modelling the PGP web-of-trust, whereby Bob wishes to introduce any public key introduced by his trusted friend, Alice. Suppose trust specification `PGP_key(name)` represents a PGP key-name introduction, which says a truster believes the PGP identifier of a subject. The PGP web-of-trust policy can then be modelled as:

```
PGP_key(name): Alice -> p asserts PGP_key(name): self -> p
```

Here we use a notational shorthand to make public keys more readable. We assume "`Alice`" expands to her real public key. Delegation of trust is purely internal to a principal. A subject may not know or even care if a trust instance is delegated, e.g. suppose Cindy learns a `PGP_key()` instance from Bob, but she may not necessarily know how Bob derives this assertion. Delegation of trust is unlike delegation of rights in that it is weaker. It does not require or force a principal to perform some action, nor does it guarantee any responsibility, where these are typical for delegated authorization [111, 148]. Delegation of trust is merely a mechanism for deriving new beliefs.

There exists a special type of trust delegation, called *blind delegation*. Blind delegation is where a principal asserts all trust instances by other principals. A possible use is for a principal acting as a *trust proxy*, e.g. a representative principal in an organization. This can be specified as,

```
any-statement: p1 -> p2
asserts any-statement: self -> p2
where p1 == 0x14ba9b925 || p1 == 0x5918b01a || ...
```

where the list of proxied principals is constrained by variable `p1`.

There are several reasons it might be desirable to set up a trust proxy. First, it provides a single identity for external parties, as the example above shows. Second, it presents a central point of management, so that only certain principals can represent the organization, e.g. those who are trusted by the proxy. Third, it provides a single principal for audit purposes. However blind delegation is usually over-general, which limits its applicability.

Example: Transitivity Trust is usually not transitive [126, 114]. That is, if A trusts B, and B trusts C, it does not automatically imply A should also trust C. However, trust *can* indeed be transitive if its context is sufficiently specific and restricted. For example, if Alice trusts Bob as her legal consultant, she may also trust other legal experts that Bob refers to. This may be because Alice is unfamiliar with legal matters and hence solely relies on Bob's advice. This could be encoded as follows,

```
legal_advisor(): self -> Bob, legal_advisor(): Bob -> p
asserts legal_advisor(): self -> p
```

Transitive trust is complementary to delegating trust. Delegation of trust allows others' beliefs to become a principal's own belief, and is determined by the subject. Transitive trust, on the other hand, allows a truster to establish trust relationships with principals its subject trusts and is controlled by the truster. Figure 3.3 contrasts transitive and delegating trust. Solid lines represent original

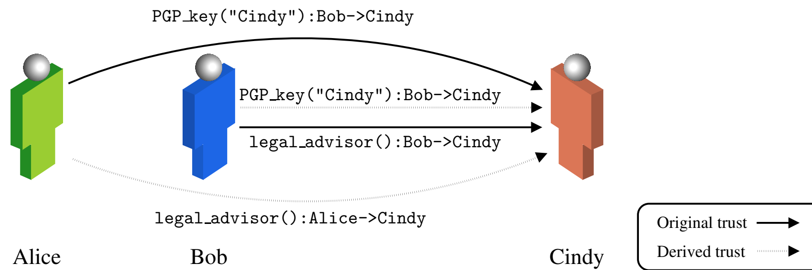


Figure 3.3: Transitive trust and delegating trust

trust relationships, and dotted lines represent derived trust relationships. The top two lines represent delegating trust, where Bob derives his assertion regarding Cindy's PGP key based on Alice's assertion. The bottom two lines represent transitivity, where Alice relies on Bob as her advisor and learns to trust Cindy as a referral advisor, based on Bob's recommendation.

3.5.6 Action policies

In the previous section, the **grants** clause allows explicit action instances to be given to a trust instance. Another approach is through *action policies*. An action policy relates action instances with trust instances, subject to conditions. The most obvious use is to express trust-based authorization, where action instances correspond naturally to access requests. Another use is to express trust decisions, where action instances represent queries that one may wish to ask. Yet another use may be to define *obligation*, i.e. actions that must be taken when certain trust is met. It is up to a principal to decide what its action policies are for.

As with trust policies, we shall first describe *action templates*, which are partially instantiated actions for the purpose of constructing new actions in action policies. Their syntax is provided below:

$$\textit{action template} ::= \langle \textit{action name} \rangle \; \textbf{(} \; \langle \textit{parameter} \rangle \textbf{,} \; ... \; \textbf{)} \; \textbf{:} \; \langle \textit{requester} \rangle \quad (3.22)$$
$$\textit{requester} ::= \langle \textit{principal specifier} \rangle \quad (3.23)$$

An action template includes a name, a list of parameters as defined in (3.20), and a *requester*, which is a principal specifier (3.15). Effectively, an action template represents an action initiated by the matching requester, with the matching parameters.

With action templates defined, it is now possible to describe the syntax for action policies:

$$\begin{aligned} \textit{action policy} ::= \; & [\; \langle \textit{trust use} \rangle \textbf{,} \; ... \;] \; [\; \textbf{without} \; \langle \textit{trust use} \rangle \; \textbf{,} \; ... \;] \\ & [\; \textbf{where} \; \langle \textit{conditions} \rangle \;] \; [\; \textbf{set} \; \langle \textit{assignments} \rangle \;] \\ & \textbf{grants} \; \langle \textit{action template} \rangle \; \textbf{,} \; ... \end{aligned} \quad (3.24)$$

The syntax for action policies is a subset of the syntax for trust policies. The main difference is that action policies mandate a **grants** clause, and do not have **asserts** and **valid** clauses. The evaluation and parameter handling semantics for action policies are consistent with trust policies.

Consider a follow-up to the example on bootstrapping trust in the previous section on Page 50. Assuming a distributed file service cooperates with the authentication service, and protects its files using an access control list (ACL) represented as a database table, `ACL`,

Field	Type	Description
object	string	Name of an object
user	string	Authenticated user identifier
mode	string	Access mode (e.g. 'read', 'write', 'modify')

It may abstract access control queries into an action template, specified as,

```
access (string obj, string mode)
```

where `obj` gives the object requested, and `mode` gives the requested access mode. The access control policy, assuming `AS` is the key for the authentication service, can then be specified as,

```
user (user_id): AS->p
where [[ SELECT *
         FROM ACL
         WHERE object='$obj' and user='$user_id' and
               mode='$mode' NOT NULL ]]
grants access(obj, mode): p
```

When a principal requests access to a file, it is first authenticated with the authentication service, which creates a `user()` trust instance. The principal may then present this `user()` instance to the file service, which constructs an `access()` action instance representing the request and then evaluates the policy for a decision.

Comparing this with the construct for authorization trust in the previous section, action policies present a separation between action and trust. There are several reasons for this separation. When a trust statement is specified, its exact uses may not be known in advance. Indeed, as a trust statement represents a belief, it is often up to the particular principal who receives it to decide how it should be interpreted and used. Second, if a principal makes access control decision based on trust, it is sensible for the principal to define its own access control policies, since the principal is taking the associated risk of breached access. This is especially true in distributed environments. A similar concept can be found in Herzberg et al. [101], but with a focus on access control.

3.5.7 Conditional and assignment expression

Expressions exist in two places in the Fidelis Policy Language, as conditions in a **where** clause, or as assignments in a **set** clause. The choice of an expression language depends highly on the application nature and complexity of the trust policies a principal wishes to express. It is deliberately left as a choice for each individual in Fidelis. The syntax described here gives a reference language used throughout this thesis. It is, however, not intended to serve all needs. For some principals, a simpler language will suffice, while for others, more advanced operators might be required.

The expression language includes seven types of operator: comparison, (Boolean) logic, numeric, string, assignment, principal and group operators. They are summarized in the following table:

Type	Operators
Comparison	`== != > >= < <=`
Logical	`&& ||`
Numeric	`+ - * / % ^`
String	`+ subst ~=`
Assignment	`=`
Principal	`== [] in <`
Group	`{} [[]]`

Most of these operators are straightforward, those which are not are explained below. The string operator `+` is for concatenating two strings into one. The string operator `~=` performs regular-expression pattern matching. The left-hand side refers to a string variable and the right-hand side specifies a regular expression. The evaluation returns a Boolean result. The syntax for regular expressions is a simplified form of those present in the Perl language. Some examples highlighting the syntax are provided in Figure 3.4. The string operator `subst` performs a substring test. For example, the expression

Expression	Matching
`foo`	A single string "foo"
`foo|bar`	A choice between "foo" and "bar"
`(foo)* bar`	Zero or more "foo" followed by "bar"
`(foo)? bar`	One or none "foo" followed by "bar"
`(foo)+ bar`	One or more "foo" followed by "bar"

Figure 3.4: Examples of regular expression patterns

```
"foo" subst "foo bar"
```

evaluates to **true**.

The principal operator `==` performs an equality test on a pair of public keys. This compares the actual keys, its algorithm and other associated key information. The unary operator `[]` takes a threshold principal and returns its threshold value. The operators `in` and `<` are group operators, which take a principal expression and a group expression. The `in` operator determines a principal's membership of a group. The `<` operator is specifically designed for threshold principals. It determines whether a variable for a threshold principal on the left-hand side is satisfied, provided the threshold group is defined on the right-hand side. For example,

```
p < 3-of { 0x521ba915, 0x1b0a06f4, 0xe89a5bc01, 0x510a0f7e4 }
```

tests whether `p` constitutes at least 3 principals in the group on the right-hand side. Note that the threshold group is specified in the syntax of (3.1).

The group operator `{ }` allows literal specification of groups, by listing the members separated by commas. The operator `[[ ]]` encloses an embedded SQL query statement. Parameter communication with an embedded SQL statement is provided through an escape character `$`. For example, `$a` in

```
[[ SELECT username FROM user_db WHERE id='$a' ]]
```

will be replaced with the value of variable `a` at evaluation. An SQL-driven group expression allows the group for a group or threshold principal to be defined dynamically by database queries. This is particularly useful if the size of a group is large, or if the definition of a group is independent of policy specification, i.e. addition or removal of group members need not rewrite the policy.

3.5.8 Evaluation semantics

We assume a principal has access to a trust policy engine, simply referred to as *policy engine*. A policy engine maintains a *trust base*, T, which consists of a set of trust and action policies, and processes queries over those policies contained in the trust base. In abstract terms, a query consists of a set of trust instances and a *query template*, which is either a trust template or an action template. A query with a trust template attempts to determine whether a trust relationship can be established, given a set of known trust instances. Similarly, a query with an action template determines whether an action can be or is to be performed, given a set of known trust instances. For the description of semantics, we shall assume the trust instances in a query have been cryptographically verified for their integrity.

A policy engine takes a query as input and returns a trust or action instance and optionally a *trace* of execution. The resulting trust or action instance is an instance that *matches* the query template. This means for an action template, that an action instance must match its name and all parameters whose values have been given in the template. For a trust template, additionally, the truster and subject principals must also match. The execution of a query consists of a sequence of *evaluations* of policies in T. Each evaluation works in the context of a single policy, and takes as input a set of trust instances, a query template and an environment, and a trust or action instance is returned as output.

Conceptually, we can represent the execution of a query as a digraph, $D = (V, E)$, where vertices are sets of trust instances and edges are trust or action policies. The goal of a query execution is to find a path in D,

$$v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} ... \xrightarrow{e_{n-1}} v_n$$

such that $t_{n-1} \in v_n$ is a trust or action instance that matches the query template, where v_1 is the set of trust instances given as part of the query. An edge represents an evaluation of a policy. Semantically, this means, for an edge $e_i = v_i \rightarrow v_{i+1}$, assuming $v_{i+1} = v_i \cup \{t_i\}$, to find a minimal subset $v'_i \subseteq v_i$ such that the evaluation of the policy p_{e_i} that takes input v'_i and some environment would output t_i. Additionally,

1. for each trust use in p_{e_i}, there exists exactly one corresponding trust instance in v'_i. Correspondence means the trust instance must be an instance of the trust use, *and* its parameters must agree with their binding, as defined below.
2. every variable must be bound to a value. For a trust use, a parameter or principal variable must be bound to a value provided by the corresponding parameter or principal in its trust instance. For a trust or action template, a parameter variable is bound to a value provided either by a previous binding, the query template or a name-value pair from the environment. Where multiple bindings are possible for the same variable, all bindings must agree to the same value.
3. for each trust use in the **without** clause of p_{e_i}, there must not exist a corresponding trust instance in v_i and in any other mandatory repository.
4. all parameter bindings must satisfy the conditional expression, i.e. must evaluate to **true**, if available.
5. if the optional assignment expression exists in p_{e_i} it must be evaluated after all variables are bound. Since evaluation of assignment expressions may create or modify variable bindings, this requirement guarantees it will not cause unexpected side-effects.
6. all trust instances in v'_i must be valid according to their validity conditions.

The resulting trust or action instance is computed by instantiating the query template, filling variables with their appropriate bindings. For a trust instance, the validity condition will be as explicitly specified, if it exists. Otherwise, it will be determined following these rules:

- If there exists a trust instance in v'_i whose validity condition is by online status check, the new validity condition will be set to the online status check. This gives the same effect as a clause of **status** validity.
- If there exist trust instances in v'_i using any of the time bounded methods (namely validity period, timed CRL or timed renewal), the new validity condition will be computed by recursively combining pairs of validity conditions until left with one. The combining algorithm for time bounds b_1 and b_2 results in b, where $b = b_1$ if $b_1.end < b_2.begin$, or $b = b_2$ if $b_2.end < b_1.begin$, or

  $$\begin{aligned} b.begin &= \max(b_1.begin, b_2.begin) \\ b.end &= \min(b_1.end, b_2.end) \end{aligned}$$

  The type will default to the offline validity, unless otherwise stated by the policy.
- If all trust instances in v'_i are permanently valid, the new validity will be permanent. This has the same effect as explicitly specifying **always** for the validity.

Let $e_i = v_i \rightarrow v_{i+1}$ and suppose v'_i satisfies e_i, then a pair (e_i, v'_i) is called a *realization* of the trust policy. The chain of realizations $((e_1, v'_1), (e_2, v'_2), ..., (e_{n-1}, v'_{n-1}))$ is called the *execution trace* for a query. The execution trace provides detailed information how the policy engine derives an answer, and may be useful later as a proof of the correctness of this answer.

3.5.9 Discussion

This section compares Fidelis with PolicyMaker [9], KeyNote [20], REFEREE [98], TrustEstablishment [101], SPKI [18] and OASIS [15, 3, 5]. We now focus our discussion on the *representation of credentials*, *expressive power*, and *validity for credentials*. As a general note, the focus of these systems differs: SPKI and OASIS are designed to facilitate distributed access control; PolicyMaker and KeyNote generalize distributed access control into the management of trusted actions; TrustEstablishment, on the other hand, with its Trust Policy Language (TPL), has a specific focus on mapping principals identified by certificates into roles, which can then be used in conjunction with existing role-based access control mechanisms. Fidelis facilitates general trust-related queries, which may or may not be related to actions. Due to these inherent differences, some aspects are not comparable among these systems.

On representation of credentials. Fidelis represents trust statements as first-order predicates which can carry typed parameters. The predicate representation allows arbitrary belief to be expressed although its interpretation is subject to the local knowledge of a principal. This may be determined by prior agreement, by standards or by automatic discovery or negotiation. From the specification point of view, the parameters of a trust instance serve as an *interface* for use in policy specification. This increases the expressiveness of a policy by exposing relevant details of a trust statement that may be of interest to policy writers. The predicate representation in Fidelis originates from OASIS, where predicates are used to represent roles, appointments and authorizations.

SPKI uses public key certificates to represent beliefs. Conceptually, a SPKI certificate is a collection of named attributes. Provided a principal may define arbitrary attributes, this representation is equally expressive as the predicate form in Fidelis. However, SPKI certificates are primarily for expressing authorization and its delegation, and sometimes for name-key binding. Using them for general beliefs is considered a "non-standard" use.

PolicyMaker, KeyNote and REFEREE represent credentials and policies (which are collectively referred to as assertions in their terminology) as programs. The idea is that the expressive power of assertions therefore matches the expressive power of the chosen programming language. REFEREE goes a step further, allowing the use of arbitrary languages, and it has a mechanism to automatically download appropriate language interpreters if needed. The approach of programmable credentials, while achieving a high degree of expressiveness, suffers complexity, maintainability and efficiency problems. Furthermore, this makes it more difficult to guarantee the correctness of a policy, which implies proving the correctness of its program. In KeyNote, credentials and policies are written in a constrained expression language. This, as its designers noted [19], is a trade-off between expressiveness and efficiency.

On expressive power. A policy in Fidelis is either a trust policy or an action policy. Trust policies are intended for general trust queries, while action policies are for action-based queries, e.g. access control. A policy specification may demand a prerequisite set of trust instances minus a set of trust instances that must not exist. In addition, Fidelis allows fine-tuning of policies based on parameters in trust instances, their trusters and subjects, and an extension mechanism for supporting application-specific semantics. Furthermore, its inclusion of group and threshold principals supports real-life policies related to multiple parties. These combined features achieve a high degree of expressive power, supporting prerequisite-based, recommendation/reputation, and delegation-based policy types. Note specially that the support for general prerequisite-based policies considerably increases its expressiveness, given that most sources of trust, as discussed in Section 3.2.3, can be captured through this mechanism. For example, requiring a recommendation from certain friends can be naturally expressed as a prerequisite condition. Also important is its ability to express negative, non-monotonic policies. This is convenient and indeed sometimes essential: if one can specify policies covering all possible aspects of a matter, then it may assume the absence of certain trust instances implies distrust. However, it is often difficult if not impossible to capture all such aspects even for a simple system,

and thus explicit distrust as an instrument to express negative assertions becomes an essential tool for guaranteeing consistency of policies.

The main type of policy that PolicyMaker, KeyNote and SPKI attempt to capture is *delegation of authority*. These systems share a similar basis for processing credentials, which aims at finding a *delegation path* from presented credentials to some trusted local policies. However, the details which affect their respective expressiveness differ. In PolicyMaker and KeyNote, credentials and policies act as filters on query strings, which return a compliance value (e.g. accept or reject). This mechanism allows complex, application-defined query strings to be evaluated. In SPKI, credentials are conceptually represented as *tuples* and are processed by an *tuple reduction* algorithm. SPKI tuple reduction is specifically for reducing chains of delegation to derive authorization decisions, and thus is not sufficiently expressive for general policies. Note that these systems may express recommendation policies, by treating recommendation orthogonally to delegation. This however has an undesirable consequence since delegation often relates to responsibility and power, while recommendation often does not. Furthermore, PolicyMaker and KeyNote only support monotonic policies for simplicity reasons. SPKI allows an extensive choice of validity methods, including revocation. This is discussed next. Another point to note is that these systems handle purely action-related queries – an influence from their origin of access control.

OASIS is for distributed role-based access control, with an extensive support for policy-driven role activation. Role activation may be subject to prerequisite roles, *appointments* and *environmental predicates*. An appointment can be considered as a special kind of trust statement, whose intention is to allow role activation. These components allow complex real-world policies relating to roles to be specified. Many ideas in Fidelis originate from the research on OASIS, especially prerequisite conditions and parameter handling. While OASIS has extensive policy support, it is not designed for general trust policies, for example, policies with recommendation or reputation are awkward for OASIS.

TrustEstablishment is similar to OASIS in that policies are used to direct role assignments. It supports recommendation-based policies, which map a collection of recommendation certificates into a role. It has filter mechanisms based on simple conditions and certificate types. It also has a mechanism for negative credentials to be verified. However, it lacks support for general prerequisite, and application-defined conditions. Moreover, its support for non-monotonic policies does not allow for fine-grained specification, given that it is simply based on a revocation list approach. While conditions on fields can be specified, it does not allow inter-certificate correlation as provided by Fidelis. This poses some limitations on its expressiveness.

On validity. Neither PolicyMaker nor KeyNote have any provision for invalidating credentials. The primary reason is due to their monotonicity, which assumes that absence of a credential or policy has a negative implication. TrustEstablishment depends on the X.509 validity semantics, which uses a validity period that may be overridden by a revocation list. OASIS opts for a validity scheme backed by asynchronous messaging for rapid revocation of credentials. This is due to its demand for a high degree of security. SPKI in its current proposal [18] has an extensive choice of validity schemes, both offline and online. Its online methods include timed CRL, revalidation, and one-time revalidation. The idea of timed CRL and timed renewal in Fidelis originates from SPKI. Nevertheless timed renewal differs slightly from timed revalidation. Timed renewal is effectively identical to automatic issuance of a new trust instance at the end of a validity period, while timed revalidation only refers to an existing credential. There are two more differences. First, Fidelis supports an online status check for situations where absolute assurance is a must; second, it has a provision for asynchronous messaging to maintain online validity – an influence from OASIS research.

3.6 Summary

Fidelis is a trust management infrastructure, based on the concept of *trust conveyance*, which models the mechanism by which a piece of trusted information propagates from one principal to another.

In Fidelis' terminology, the trusted piece of information is referred to as a *trust statement*, which is typically an assertion held by a principal regarding some other principal. The principal making a trust statement is the *truster*, and the principal to which a trust statement is related is the *subject*. The principal who sends a trust statement is a *conveyance source* or just a *source*, and the principal who receives a trust statement is a *conveyance target* or just a *target*. It is important to note that neither a source nor a target are required to trust the relevant trust statement; they are just participants in a trust conveyance. It is the *trust policies* that determine if a trust statement is trustworthy.

While not a strict requirement for participating in a conveyance network, it is advantageous to have a common language for the specification of trust statements and their relationships. A language has been developed and presented for this purpose: the Fidelis Policy Language (FPL). This language refers to the structure of a trust statement as a *trust statement specification* or *trust specification*, and a concrete instance of it as a *trust statement instance*, or *trust instance* for short. A trust statement is modelled as a predicate with typed parameters. A trust instance also has an explicit truster and subject, which may be either a simple principal, a group principal or a threshold principal. The language defines a syntax and semantics for specifying two types of policies: trust policies and action policies. A trust policy defines a trust relationship that may be subject to: (1) prerequisite trust instances, (2) absence of certain negative trust instances (i.e. distrust), (3) conditions on parameters in trust instances or principals. An action policy relates action and trust. It embodies action-related trust queries, e.g. authorization. In Fidelis, a trust instance has a validity condition, which may be expressed either as a validity period, or using one of the online means, including *timed CRLs*, *timed renewals* and *status checks*.

In the next chapter, a web service architecture for Fidelis is described.

4 Fidelis and Web Services

Over the past decade, interest in distributed computing has led to the development of several middleware platforms. Among the most influential are the Distributed Component Object Model (DCOM), the Common Object Request Broker Architecture (CORBA), and more recently, Java Remote Method Invocations (RMI) and Jini. These platforms provide a Remote Procedure Call (RPC) mechanism, and usually a set of platform services to support distributed processing, such as naming, trading, transaction, security, etc. However, none of them has succeeded in establishing itself as the universal standard.

The emergence of web services represents a step towards a unifying middleware platform. This chapter describes the design and implementation of Fidelis on the web service platform. Section 4.1 provides an overview of web services and discusses the design issues of implementing Fidelis as web services, with a focus on interoperability and communication with unfamiliar parties. Section 4.2 describes its architecture which consists of a collection of *nodes* implementing interfaces. This section describes the interfaces that facilitate trust management. Section 4.3 and 4.4 address the issues of data representation. Section 4.3 describes an interchange format for policies that is designed to enable interoperability between heterogeneous principals. Section 4.4 describes an XML-based (Extensible Markup Language) [149] format for representing credentials, called the Fidelis Interoperable Credential (FIC) format. FIC serves as a common representation for the exchange of Fidelis trust instances in open web services.

4.1 Introduction

In this section, we first provide a brief overview of the web service platform and its constituent technologies. At the time of writing, web service technologies are yet to be fully standardized, and many are still under extensive research and development. The three pieces of technology introduced here, namely Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI) are the de-facto standards in the industry with some widespread use, and are promising to be accepted as formal standards.

After this introduction, a discussion on various design issues for implementing Fidelis based on web services will be presented. The focus of the discussion will be on the impact of the open and global nature of the web-service architecture.

4.1.1 Background

Web services are built on top of web technologies. The central notion is the ubiquitous use of XML (Extensible Markup Language) [149], e.g. for message representation, definition of remote interfaces, and description of interaction. The platform consists of three main components: the Simple Object Access Protocol (SOAP), the Web Service Description Language (WSDL), and the Universal

Figure 4.1: A sample SOAP message (message content from [7])

Description, Discovery and Integration Service (UDDI). Nevertheless there are additional services being actively worked on, e.g. for business process modelling (ebXML [150, 151]); for security support (WS-Security [152], XKMS (XML Key Management Specification) [153], SAML (Security Assertion Markup Language) [154, 155]), etc.

SOAP [7, 156, 157] is the fundamental messaging technology for web services. It is an XML-based protocol, defining a standard representation for XML messages, the processing semantics and the encoding of typed data. The standard specifies a mechanism for utilizing the protocol to facilitate RPC-style invocation over HTTP, and a one-way message passing mechanism over SMTP. A SOAP message consists of two parts: a body block and an optional header block. Both header and body blocks may contain one or more elements (called *information items* in XML terminology). The body block serves as the container for the message, while the header block is intended for extensions. A SOAP message may be processed in a pipeline of SOAP *nodes*. Each node may be designated to handle certain extensions. An extension in the header may be declared as mandatory, in which case it must be processed along the pipeline. The body may contain any XML document, in particular, a representation for a remote invocation. The standard also specifies a special body payload for exception conditions, referred to as *faults*. Figure 4.1 gives a sample message, with its parts highlighted. Its header contains a mandatory extension `Transaction`, and its body shows the response of an invocation to method `GetLastTradePrice`.

SOAP deals with the low-level packaging of messages. WSDL [158] addresses the next layer up – namely description of remote services. The description covers two areas: the specification of services and deployment information. The service specification consists of a collection of *operations*. Every operation specifies its input and output *messages*. An operation may be one-way, request-response, solicit-response or notification, depending on the existence of input and output messages. A message may include a collection of typed *parameters*. *Types* may be specified in XML/Schema [159, 160] or other schema languages. The deployment information for a service is specified as a collection of *ports*, where each port offers a set of operations. A concrete binding for a service specifies the URL address for those ports. A feature in WSDL is that components are separated into abstract definitions and concrete bindings, which allows reuse of components. For example, an abstract message definition may be bound into two different data representations, one as a SOAP message and the other as an HTML form submission.

UDDI [161, 162] complements WSDL, providing a registration and discovery framework for web services, i.e. a trader service. Conceptually, it offers three types of information: white pages, containing the contact details about a business; yellow pages, also containing the contact details but organised via a classification taxonomy; and green pages, containing the technical information for accessing

the services. The white and yellow pages information is represented in a *businessEntity* structure. It is associated with one or more *businessService* structures, which describe the services or business processes offered, with some optional, human-readable description. The green pages information is described in a *bindingTemplate*, which is associated with every businessService. It contains two vital pieces of information, an access point and a binding key to a *tModel*. A tModel serves as an abstract standard, defining the service behaviour and the wire protocol (possibly in WSDL).

4.1.2 Design issues

One of the fundamental characteristics of the web service environment is its global and open nature. Any application design for web services must therefore consider the circumstances where previously unknown principals attempt to interact. Existing technologies such as the UDDI offer solutions at the service level, i.e. searching, locating and invocation of services. Issues specific to trust management must also be addressed. Prior to the discussion on these issues, recall that principal autonomy is one of the prime principles behind the design of Fidelis, as described in Chapter 3. This means, broadly, that a principal has the discretionary power to:

- define its policies. This includes both the definition of trust statements, actions and their inter-relationships.
- decide the means to define and describe its policies. The use of the Fidelis Policy Language is one possibility, while other possibilities include using a proprietary language, or some graphical policy editor.
- choose appropriate data representations for its trust instances. Choices may be subject to internal interoperability, backward compatibility, or technologies available.

With the notion of principal autonomy in mind, some major design issues can be described:

1. **Interoperability.** When two principals (either previously known or unknown to each other) attempt to interact, these issues must be considered:

   1.1 **Credential representation.** Credentials in the system are trust instances. The design should allow a variety of representations for trust instances, e.g. X.509 or SPKI certificates, XML credentials, etc. A common representation must be agreed upon by both sides or, under some circumstance, it may be sufficient for the recipient to understand the sender's credentials.

   1.2 **Common ontology.** As trust statements are intended to express arbitrary beliefs, it is essential to establish a common vocabulary (ontology) that specifies the structure and semantics for trust statements.

   1.3 **Policy representation.** If principals need to exchange policies (see item 2 and 3), a common representation for policy exchange must be agreed upon.

   1.4 **Policy semantics.** As policies may be specified by different means (e.g. policy languages or tools), different semantics exist. Establishing a common policy framework is hence a prerequisite to enable policy-level interoperation.

2. **Policy discovery.** Assuming a principal discovers other unknown principals through some dynamic discovery scheme such as UDDI, it will further need to find out the policies supported by these principals in order to gain trust (i.e. obtain trust instances) or request services.
3. **Policy negotiation.** More advanced principals may support policy negotiation, which gradually works towards an agreement with unknown parties, by incrementally disclosing and exchanging policy and credential information.
4. **Credential disclosure.** Provided the policy is known, it is often desirable for a principal to disclose the least set of credentials, just sufficient to satisfy its request. This prevents information leakage through over-disclosure.

5. **Lightweight principals.** The design should have provision for lightweight, mobile principals. There are two sub-issues:

   5.1 **Credential management.** Mobile devices tend to be small, limited in resources and more exposed to security hazards. One option is to delegate the tasks of credential management to other principals where appropriate, thus reducing the use of resources on the device, and at the same time preventing credential or key theft.

   5.2 **Support for disconnection.** A disconnected principal should not cause disruption of the conveyance network in which it has participated. In particular, the disconnection of a truster should not prevent the use of trust instances it has issued. Symmetrically, a disconnection should cause a minimal impact on the usual operations of the disconnected principal.

The issues discussed here drive the design decisions throughout the development of the work presented in this chapter. Where appropriate, references to these issues will be made in the rest of the chapter.

4.2 Service architecture

The system consists of a collection of *SOAP nodes*, as defined in [156]. A SOAP node is a processing entity for SOAP messages, and may generate messages for other SOAP nodes. Each node may provide services as methods. These methods are mapped directly into Fidelis actions, where the method name maps to the action name, and arguments of a method invocation map to parameters of an action instance. A node may also implement a number of interfaces to support trust management services, in addition to its own methods. These interfaces are defined in WSDL, and include *conveyance*, *trust inference*, *credential management*, *policy interrogation*, and *trust agent*. These are collectively referred to as the *Fidelis interfaces*.

4.2.1 Locating principals

A SOAP node may represent one or more principals. By this, we mean that a node may implement interfaces on behalf of principals, primarily for two purposes:

- *credential management*, which includes conveying trust, managing and safeguarding trust instances;
- *trust inference*, e.g. interpreting and answering queries against the principal's policies.

A node maintains a list of principal identifiers that it represents. There is no strict mapping requirement between principals and nodes. A node may represent a single principal, or may be shared among multiple principals – likely in an organization. It is also possible for the same principal to be represented by multiple nodes, e.g. a user on the move may simultaneously be represented by both her mobile device and her office computer.

A problem that needs to be addressed is the location of principals: given a principal identifier, find the list of nodes that act on its behalf. Before the discussion of possible solutions, it is worth noting that this lookup is required if principals only know each other by identifiers. An example is where a principal intends to convey trust instances to a friend, in which case the node where the friend is represented needs to be discovered. Communication between strangers often starts by contacting a node, either previously known, or located dynamically by UDDI.

We refer to an instance of node-principal binding as a *presence*. A presence can be discovered through a number of means. The architecture does not prescribe a standard approach but instead leverages existing web service technologies. A presence may be directly bound to a principal identifier, e.g. the truster field of a trust instance may include an attribute that gives the URLs of representative nodes. However, this solution is only possible if the presence is static. Therefore it is more suitable for principals with a well-known, persistent presence, e.g. a University, a government agency, etc. For

individuals whose presence frequently changes, ad-hoc, out-of-band solutions such as e-mail communication may suffice. A more plausible approach, however, is to employ directory services. A principal identifier may hence be associated with a list of URLs of directory services, where the current presence of the principal may be looked up. A URL format for referencing LDAP entries is described in [163] and may be used for this purpose.

Another, more web-service centric approach is to register principal identifiers with UDDI registries, as an entry in the identifierBag of a businessEntity structure. The principal representing a business together with the binding location can then be searched using standard UDDI methods. One could also host a white pages, directory service for a local domain (e.g. a department, a branch, etc), mapping principals to nodes and vice versa.

4.2.2 Conveyance interface

The conveyance interface defines the mechanisms for trust conveyance, supporting pointwise transfer of trust instances. It defines two styles of interfaces: push and pull. For the push interface, the conveyance source initiates the transfer, while for the pull interface, the conveyance target requests certain trust instances. A node may offer either or both styles. The push-style interface is suitable for a principal to actively distribute trust knowledge as it is gained, whereas the pull-style interface is suitable for a principal to passively share trust knowledge.

The pull interface defines a *getTrustInstance* method that takes a source identifier, a target identifier and a *trust template*[1]. A trust template can be thought of as a trust instance with unfilled parameters and/or truster and subject. We say a trust template is *complete* if all parameters and both the truster and subject are provided. It is different from a trust instance because it is not signed. A trust template follows the standard representation described in Section 4.4. When invoked, the node first determines whether it represents the source principal. If so, it returns the trust instances matching the trust template. The target identifier is not directly used, but gives supplementary information that may be useful for audit or security purposes, e.g. a source may refuse interaction with, or restrict interaction to, certain principals.

For the push interface, a source sends trust instances asynchronously. The target principal first *registers* for conveyance, expressing its interest in certain trust instances. A registration request consists of a source identifier, a target identifier, a trust template, reply addresses of the target principal and a *registration policy*. Once a registration is received, the node determines if the requested conveyance is allowed and raises an exception if not. Otherwise, it adds the registration into a registration table, which contains entries of registration, indexed by the source principal and the name of trust instances. When the node learns about a new trust instance owned by principals it represents, it checks through the table and initiates the conveyance process according to the policy of each registration if matches are found. A node learns about new trust instances from a number of sources: directly from those principals it represents, conveyance from other principals, or as results of trust inference, see Section 4.2.3.

The registration policy is a collection of name-value pairs, expressed in an XML format. It fine-tunes how a registration is handled. A node should respect the registration policies it understands, and may discard those it does not. A registration policy may be tagged as mandatory, similar to SOAP header entries, in which case, registration will fail if the node fails to understand or comply with the policy. This processing semantics allows application-specific, extended policies to be supported. The standard policies are summarized:

- **Urgency** A target principal may express how soon a new trust instance should be sent. Possible choices are: *immediate*, *bounded time*, *bounded volume* and *unspecified*. If *immediate* is specified and the policy accepted, the node should attempt, by best-effort, to send new trust instances as soon as it knows about them. *Bounded time* specifies a maximum period of time a node can hold a trust instance without attempting to send it. This period excludes delay caused by network

[1] Note that *trust template* here is not the *trust template* syntax term in Chapter 3 although it serves a similar purpose.

failures. *Bounded volume* allows the target principal to specify the maximum volume of trust instances to receive over a fixed period, e.g. maximum 100 per hour. This is useful to reduce the resource overhead by constraining the receiving rate of new trust instances. *Unspecified*, which is the default, allows a node to choose the most convenient time to commence trust conveyance. This policy allows a target principal and a source node to trade off between trust urgency and resource load. It serves mostly as a hint rather than a strict demand.

- **Persistence** By default, a registration entry is removed once a conveyance is successfully completed. If a trust instance invalidates frequently, a repeated registration may be requested. It is qualified with either the number of repetitions or an expiry time. This determines when a registration entry can be removed.
- **Reliability** If there is a network fault when a conveyance is taking place, the process will fail without further retry. This behaviour is acceptable if the node also supports a pull interface or has other means to deliver trust instances. A registration may demand reliability, in which case a failed attempt will be queued and retried at a later time, until it succeeds or a threshold number of times has been tried.

Note that the processing of certain registration policies may require the source principal to maintain states about the interested target principals. For example, the semantics of the *bounded volume* policy requires the source to remember the number of trust instances sent to a target principal, and that state needs to be reset periodically. The mechanism to support asynchronous trust conveyance (i.e. the push interface) is therefore stateful. However, this is different from the mechanism for processing trust instances, which is stateless. This will be further clarified in the next section.

4.2.3 The trust inference interface

The trust inference interface encapsulates the evaluation of both trust policies and action policies. It has a single method, *infer*. For space and readability reasons, its definition is given here in Java syntax. However, note that the actual interface definition is in WSDL.

```
InfResult infer (CredentialSet trust_instances,
                 QueryTemplate template,
                 Environment environment,
                 int flag)
```

The argument `trust_instances` is a set of trust instances (or their references), `template` is either a trust or action template, `environment` is a set of name-value bindings, and `flag` specifies additional settings, e.g. whether the inference trace should be generated. The return value contains either a (possibly empty) set of new trust instances or a Boolean value, and optionally an inference trace. The evaluation of this method depends *solely* on the given arguments, which are either provided by the requester (`trust_instances`, `template` and `flag`) or collected from the context (`environment`). There is no state maintained between invocations. The provided mechanism for inferring trust decisions is therefore *stateless*.

The concrete representation of a trust template is described in Section 4.4. In abstract terms, it can be written as:

$$name\,(param, ...) : truster \rightarrow subject$$

where *name* gives the name of a trust statement or a wildcard, *param* may either be a value or a variable, *truster* and *subject* may either be a principal identifier, a variable or a wildcard. When *name* is a wildcard, *param* will become irrelevant. An action template can be written similarly as:

$$name\,(param, ...) : requester$$

where *name* must refer to the name of an action (no wildcard allowed), *param* may either be a value or a variable, and *requester* may be a principal identifier, a variable or a wildcard. Depending on the information in a template, *infer* answers six types of query:

- **Trust establishment** – given *name*, *truster* and *subject*, determine if the named trust statement could be instantiated. If successful, return parameter values of the result trust instance.
- **Horizontal coverage** – given *name* and *truster*, determine the complete set of principals who may obtain trust statement *name* from the specified truster, and the parameter values for the result trust instances.
- **Vertical coverage** – given either *truster* or *truster*, or both, compute the complete set of trust instances between them. This determines the maximum trust that a truster can assert (or a subject can obtain) at the time the inference is executed.
- **Complete coverage** – given a blank trust template (i.e. *name*, *truster* and *subject* are all wildcards), compute the complete set of trust instances the policies may give.
- **Action decision** – given *name* and *param*, determine whether it can be satisfied. This is the typical type of query for determining access control decisions, and is also called *authorization.*
- **Action coverage** – given *name*, determine the complete set of action instances the given set of trust instances satisfy.

When invoked, the node consults its policies and attempts to infers the required type of answer. Some policies may include methods to obtain additional environmental information, in which case, these methods are performed to complete the environment. Only if all required environment bindings are available, can a policy be evaluated. An algorithm implementing the evaluation semantics of policies in the Fidelis Policy Language is described in Section 5.1. New trust instances returned as a result are unsigned, and should be signed by appropriate nodes if they are to be used externally. When multiple trust instances are returned (e.g. for coverage queries), they will be packed into a SOAP array.

The feasibility of coverage queries depends greatly on the nature of the policies because policy evaluation may depend on environment bindings that are not provided in a query. For this reason, action coverage queries are often meaningless since the potential set of environments for these queries is often large. Moreover, if a policy has some external dependency (e.g. database queries), the result will only be valid at the instant when the query is processed since external conditions may change. The result is therefore only reliable as a hint. Furthermore, coverage queries are usually very expensive, involving inference over a large number of policies. However, if policies are free of extra environments and/or external dependencies, coverage queries enable simultaneous establishment of multiple trust relationships, and in the extreme case, obtaining maximum trust from a principal. A node should weigh the trade-offs and prudently offer coverage queries where they fit.

A trust instance passed to *infer* may either be an actual instance or a reference to locate it. A mechanism for automated credential collection is described in the next section. The method argument `flag` may indicate if the caller wishes to obtain the inference trace. However, the processing node may refuse or limit how much of the trace should be provided. It may pose a security risk if the complete trace is fed back since it enables probing of internal policies. Nevertheless, the inference trace is valuable for auditing purposes, especially if an inference node is only used internally.

A node supporting the trust inference interface is associated with a set of trust policies. These could either be directly provided to the node (e.g. loaded from a file) at deployment, or be retrieved from a *policy interrogation* node, see Section 4.2.5. This is suitable for environments where policies are to be shared among several nodes, and central policy management is desirable. Modification and update to policies therefore only take place at a single location, which helps improve policy consistency.

Figure 4.2 illustrates the interaction and relationship between a principal (i.e. the application service) and supporting components. Each "mushroom box" (square box with attached circles) represents a component, which may be an integrated software module or a separate SOAP node. Each mushroom (a circle with a line) represents an interface supported by the component. The application service receives an invocation request, upon which it consults its action policies for an access decision. It initiates a query by passing the trust instances received with the request to the inference component, where policy computation is performed. The result is then provided back to the service. The inference component is associated with a policy management component, which serves policies to the inference component upon request.

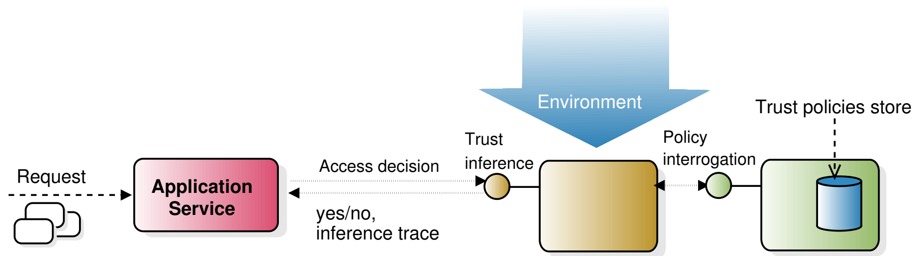


Figure 4.2: Trust inference - action decision

4.2.4 The credential management interface

Typically a principal would manage its own trust instances. Under some circumstances, it is desirable to delegate these tasks to another trusted node that offers the *credential management interface*. Some examples demonstrating this need include:

- a mobile user frequently works on different computers, e.g. at home, at the office, at clients' offices, or on the road. By keeping her credentials at a central credential management facility, it allows the use of a single identity across all these locations, yet maintaining consistency for other principals.
- for principals on mobile devices (e.g. mobile phones, personal digital assistants (PDAs), etc), because of constrained resources, greater exposure to security attacks and possible disconnections, it may be preferred to offload the credential management tasks to other trusted nodes.
- Increased redundancy. A principal may create multiple presences to improve availability. This is especially important for global networks like the Internet where faults are always occurring in some parts of the network.

Conceptually, a credential management node maintains a collection of credential bags, where each bag contains credentials owned by a principal. The credential management interface offers two categories of methods: privileged and public. Privileged methods can only be invoked by the owners, while public methods are available to anyone. A request to a privileged method needs to be signed by the requester. The node, upon receiving the request, needs to determine its integrity and freshness, and whether the requester is permitted for the requested method.

There are four privileged methods: *addCredential*, *removeCredential*, *getMatchedCredentials*, *getAllCredentials*. These methods are defined as follows (also in the Java syntax):

```
CredentialRef addCredential (Credential trust_instance)
void          removeCredential (CredentialRef ref)
CredentialSet getMatchedCredentials (QueryTemplate template)
CredentialSet getAllCredentials ()
```

All these methods identify the credential bag to operate on using the requester's identity, and as invocation of these methods must be signed by the requester, the requester identity is always known. It is hence unnecessary to explicitly add an argument to these methods to identify the requester.

The *addCredential* method adds a credential to the bag belonging to the requester and returns a reference key. The *removeCredential* method is used to remove the credential referenced by the key given as an argument from the requester's bag. Both *getMatchedCredentials* and *getAllCredentials* return multiple credentials of the requester. The *getAllCredentials* method returns all credentials belonging to the requester. The *getMatchedCredentials* method takes a trust template as argument, and returns all credentials matched by the template. This allows selective retrieval of credentials, e.g. issued by a particular truster, designated to a particular subject, or a specific trust instance with a

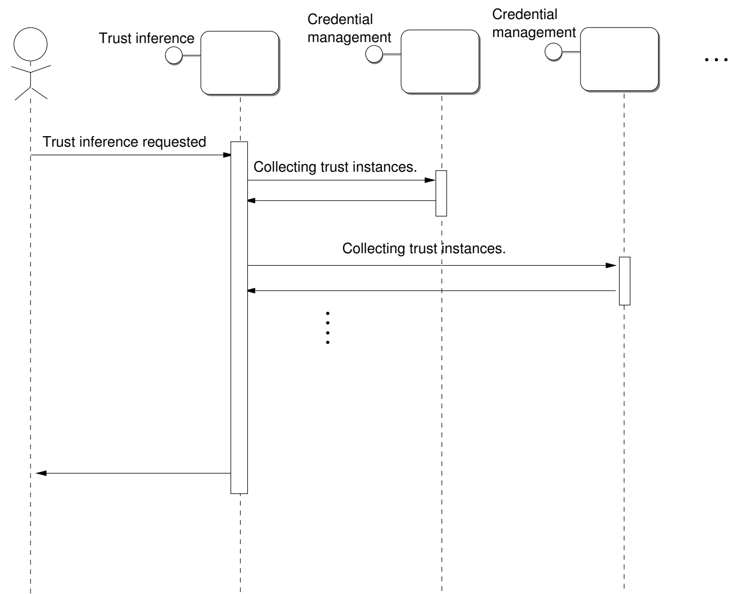


Figure 4.3: Automated credential collection

matching name and parameters. These two methods are privileged in order to prevent “credential harvest”, i.e. arbitrary retrieval by a random principal.

Public methods include *getCredential* and *getCredentials*, defined below:

```
Credential     getCredential (CredentialRef ref)
CredeentialSet getCredentials (CredentialRefSet ref_set)
```

The *getCredential* method takes a single reference key and returns the credential, and the *getCredentials* method works similarly but for a set of credentials. The security of these methods lies in the quality of the reference keys. While the format for keys is implementation-specific, the keys should not be predictable, e.g. sequential, to prevent credential retrieval based on key guessing. The recommended approach for producing the keys is to use cryptographic hash algorithms on the credentials, e.g. MD5 or SHA-1, which will provide appealing uniqueness and unpredictability properties.

The credential management interface is designed to facilitate *automated credential collection*. Recall from the previous section that trust instances passed to an inference interface may either be concrete instances or references. A reference consists of a pair of URL and key, where the URL addresses a credential management node and the key is the local reference at the node to locate the trust instance. A compliant trust inference node automatically fetches trust instances using *getCredential* and/or *getCredentials* prior to performing the inference.

Figure 4.3 illustrates the credential collection mechanism. The principal initiates a trust inference request, which causes the responder node to initiate further requests on other nodes to retrieve referenced trust instances. While the figure shows a sequential interaction, multiple credential collections may proceed simultaneously. If the fetching of credentials fails, the inference may either terminate with an exception, or continue without the missing credentials. This is determined by policy.

4.2.5 The policy interrogation interface

The policy interrogation interface specifies methods for querying and retrieving policies. It is designed to facilitate communication between strangers and enable centralized management of policies. If a principal locates a stranger with whom it wishes to communicate (e.g. to carry out a business

transaction, to obtain services, etc), one prerequisite is to find out and agree on the policies defined by the stranger party. There are two approaches to achieve this:

- A node may publish its ontology and policies. This could either be distributed at some well-known location (e.g. listed by service directories, at a public, searchable URL) or retrieved directly from the node, provided it supports a retrieval interface. The policy document should be described as a *Fidelis Policy Interchange* (FPI) document, which has an XML-structured format. The details are described in Section 4.3.
- Alternatively, a node may support programming interfaces for interrogating and discovering its policies by supporting the interface described in this section. This approach provides an opportunity for automating the process of communication establishment between strangers. This will be explained later in the section.

Comparing the two approaches, the former is suitable where some authority hierarchy exists, e.g. the top-level authority may publish a standard set of policies for subsidiaries to implement. The latter, on the other hand, is much more dynamic. It allows strangers to gain understanding and form trust relationships at runtime. This therefore requires more runtime and deployment support. It also enables centralized policy management. Centralized policy management is particularly suitable in two situations:

- For an organization, policies often tend to be large and complex. Centralized management helps reduce administrative burdens and errors because policies can be specified and analyzed at a single location, in a consistent manner.
- For mobile computing, where resources are scarce and constrained, managing policies on a separate, perhaps non-mobile, node helps reduce storage and bandwidth usage.

The principle behind centralized policy management is to separate the management tasks of policies from their enforcement. The management tasks we focus on are the storage and retrieval of policies and metadata.

The policy specification framework supported by the interrogation interface is based on the Fidelis Policy Language. Recall that policies include trust policies or action policies. We use the term *metadata* to refer to the specification of trust statements and actions. The method *getTrustSpec* and *getActionSpec* both take a name, and retrieve the specification of the named trust statement and action respectively. The specification is given as a fragment of a Fidelis Policy Interchange document. For example, suppose `T1` is declared as

```
T1 (string a, float b)
```

in the Fidelis Policy Language. The equivalent declaration in FPI would be:

```
<Statement name="T1">
  <Parameter name="a" type="xsd:string" />
  <Parameter name="b" type="xsd:float" />
</Statement>
```

This fragment creates a trust statement type which may be referenced by the policy specification returned by method *getTrustPolicies* and *getActionPolicies*. The method *getTrustPolicies* and *getActionPolicies* respectively take a trust template and an action template as argument, and return a set of policies matching the template. The rule for determining the relevance of a policy with regard to a template is based on static matching. For trust policies, the template is matched against the

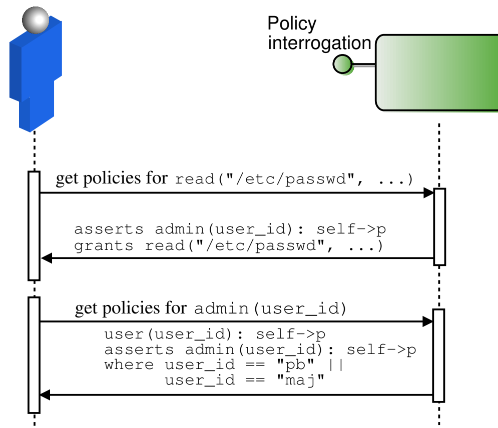


Figure 4.4: Policy discovery

trust template in the **asserts** clause; for action policies, the action template in the **grants** clause is matched. Assuming a node defines these action policies,

```
... grants read (path, ...)                (4.1)
... grants read ("/etc", ...)              (4.2)
... grants read ("/etc/passwd", ...)       (4.3)
```

Suppose a requester requests action policies for `read("/etc", ...)`, both policy 4.1 and 4.2 will be returned. Policy 4.2 matches directly with the template, while 4.1 is defined over an arbitrary parameter, which `/etc` satisfies. Policy 4.3, on the other hand, does not match the template, and is thus irrelevant. The representation for policies returned by these methods is a fragment of FPI. Please see Section 4.3.5 for details and examples.

One design goal for the interrogation interface is to support incremental discovery of policies. A requester may repeatedly interrogate a node, refining the policies to the desirable granularity. Figure 4.4 illustrates this process. In this figure, the principal first obtains the policy for the `read("/etc/passwd", ...)` action, which requires a trust instance proving to be an administrator. It subsequently queries to find out how to become an administrator. This process of incremental discovery can also be automated. This is supported through the *trust agent interface*, described in the next section.

4.2.6 The trust agent interface

As previously mentioned, when strangers make contact, there are several issues to be resolved, e.g. unknown policies and credential ontology, limited mutual trust, etc. Even when policies are known, it is still in the interest of a requester to disclose the least trust instances for a request, especially if they contain sensitive information. The trust agent interface is designed to encapsulate a principal, providing an *active* interface on behalf of the principal. It automates the process of policy interrogation and negotiation, and computes the disclosure set of credentials for requests. It supports the use of *meta-policies* to control and constrain the automation. For example, a principal may specify that certain trust instances should never be used for action policies. Meta-policies are just like other policies and may also be expressed in the Fidelis Policy Language. However, unlike other types of policy which are about trust relationships or actions, meta-policies are about policies.

A trust agent may provide assistance on several aspects. It may act as a front-end for a principal, providing a high-level interface for services. In this configuration, the principal delegates the task of selecting trust instances to the trust agent, and it issues a high-level request for service without

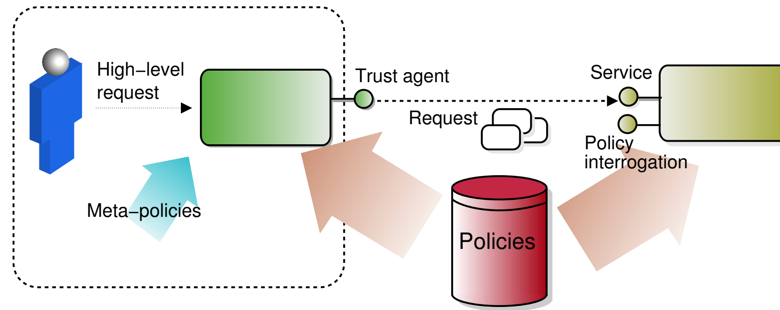


Figure 4.5: Assisted request initiation through a trust agent.

attaching trust instances to the trust agent. The trust agent then examines the action policies, selects trust instances and finally issues the actual request with those trust instances to the service node. Figure 4.5 illustrates this. In this configuration, the trust agent needs to have the private key of the principal so that it can produce requests that appear to originate from the principal. Note that the sharing of the private key implies the trust agent must be under complete control and trust of the principal. A possible implementation model is as an operating system process running as a privileged user. The trust agent also needs to have access to the credential collection of the principal. This could be achieved either by associating it with a credential management node, or implementing a custom credential management facility directly. In the former case, since the trust agent possesses the principal's key, it would be able to invoke privileged methods, thus having full access to the credential collection.

The trust agent must know the action policies for the request in advance. In the figure, the policies are published at some location that both the principal and the service can access. This is practical for cases such as reference policies published by a standardization organization. A service node may also provide its policies directly upon request, or it may support the policy interrogation interface, in which case, a trust agent can incrementally discover policies as described in the previous section. A trust agent may also consult its cached policies from the past. However, as policies may evolve, it must implement some strategy to keep its cache up-to-date. A lazy strategy would be using the cache as a hint and obtaining updates when the policies fail to satisfy requests. Caching policies is only feasible for *static* policies, i.e. those that do not depend on environments to evaluate. For *live* policies, they must be queried dynamically, tailored to each request.

One aspect of meta-policies is to allow principals to dictate the rules for choosing credentials for requests. A FPL profile for meta-policies is described in Section 5.2.3. Here we provide a brief description to motivate the approach. The profile is designed to express four types of conditions: designated principal disclosure, context-specific disclosure, trust-directed disclosure, and mutual exclusion. A principal may extend this profile or use other proprietary policies to express its meta-policies if needed. An example meta-policy is given here,

```
negotiator(): self -> 0xb258d29f
grants disclose(T2(a, b): self->p)
```

It states that trust instances matching `T2(a, b): self->p` may be used when negotiating with principal `0xb258d29f`. The meta-policy profile employs a *denial-by-default* policy, i.e. if a trust instance is not explicitly allowed to be disclosed, disclosure will be prohibited.

Trust agents can also automate *negotiated requests*. A negotiated request is an approach to mechanize the policy negotiation process. The idea of negotiated request is that a pair of trust agents carries out a negotiation conversation, gradually exchanging trust instances. When sufficient trust is gained on both sides, the request will be performed.

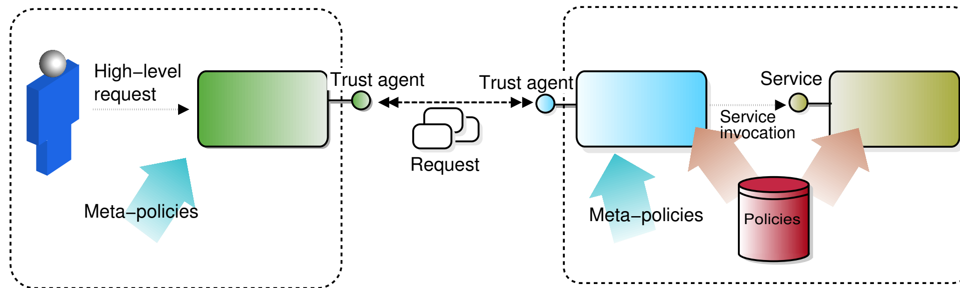


Figure 4.6: Trust negotiation between principals.

Figure 4.6 illustrates the process of negotiated requests. In this example, the requester has no direct access to the service policies, e.g. the policies might be confidential or highly dynamic therefore not worth publishing. The trust agent on the service node acts as an interceptor for the service. It interprets the meta-policies to determine whether certain policies are applicable for a request. Initially, the trust agent on the requester node issues a request with an empty bag of trust instances. The service trust agent responds with an "insufficient trust" exception, and may offer some service policies. The requester trust agent, upon receiving the exception, analyzes the offered policies with respect to its own meta-policies, and supplies more trust instances to fulfill the requirement. It may also respond with an "alternative policy" request with some credentials if it does not wish to comply with the returned service policies. Please refer to Section 5.2 for an in-depth description of trust negotiation.

Note that meta-policies in a negotiated request session play two different roles. On the requester side, meta-policies are used to specify what credentials may be disclosed and their conditions. On the responder side, in addition, meta-policies specify conditions for disclosing policies.

4.2.7 Identifying requesters

A basic requirement in any authorization system is that the requester of an invocation must be identified before authorization decisions can be made. There is no exception with Fidelis. Recall that in Fidelis, all principals are identified by public keys. This serves as a ready mechanism for establishing the requester identity. The basic idea is that a requester should sign its invocation request with its public key. A node should then first verify the signature in a request and ensure its freshness to prevent replays, and proceed if and only if these constraints are satisfied. For the work in this thesis, a solution based on SSL/TLS is designed and implemented.

SSL/TLS [164] is the de-facto standard for providing security for today's web applications. Based on X.509 certificates, the SSL/TLS protocol provides confidentiality, authentication, integrity and non-repudiation to any transport layer protocol, including HTTP – the backbone transport protocol of the WWW. For Fidelis utilisation of SSL/TLS, a principal must first produce an X.509 certificate containing its public key. Since the only relevant information in the certificate is the public key, the implementation forces the SSL/TLS stack to ignore other components in the certificate, such as the subject name, issuer name, validity period, etc. The certificate must be self-signed as a certificate chain will have a special meaning, discussed later. The SSL/TLS protocol is configured to provide at least authentication and integrity guarantees. This requires both sides of the communication to carry out a challenge-response handshake to ensure possession of the corresponding private keys. Therefore, once a SSL/TLS session is successfully established, the requester identity is also determined as a result.

Our design implements non-standard semantics in order to express requests made by a group or threshold principal. An invocation request initiated by a group principal is conceptually one that is signed by all the principals in the group (or for a threshold principal, a threshold number of principals in the group). However, SSL/TLS allows at most one certificate on each side of the communication to

be used for establishing a session. To overcome this limitation, it is necessary to interpret the semantics of *certificate chains* differently within Fidelis web services. Certificate chains are interpreted as the explicit consent of all signing principals to act for an invocation. Note that this interpretation is drastically different from the standard X.509 semantics, where a certificate chain represents a chain of certification.

Ideally, a pure XML-based solution should be developed, and some custom protocol should be designed to facilitate the use of multiple public keys in an invocation request. WS-Security [152] provides a foundation building block for adding security information, e.g. digital signatures, to SOAP messages. A recently proposed standard, the Security Assertion Markup Language (SAML) [154, 155], provides protocols which may be used to implement the semantics required by Fidelis. These developments are nevertheless left as future work.

4.3 Fidelis Policy Interchange

Fidelis Policy Interchange (FPI) is an XML document format designed for describing ontologies and policies in the trust framework. It facilitates interoperability between nodes, where internal, local policy representations may be used. The goal is to establish an interchange representation from and to which internal representations may be translated. This section describes the features of FPI.

4.3.1 Overview

FPI is based on the policy framework of the Fidelis Policy Language presented in Section 3.5, supporting both trust policies and action policies. It augments the basic language framework with XML Schema for describing types and trust ontologies, and XML Signature, for standard encoding of principal identifiers. It also integrates support for namespace management, where definitions of trust statements and actions may be qualified in declarative namespaces. In the current version, FPI documents are scoped under the namespace identifier:

`urn://opera.cl.cam.ac.uk/fidelis/FPI/04112001`

The basic structure of a FPI document consists of five parts ('*' denotes zero or more occurrences of the component),

```
<Interchange>
  <Import/>*
  <Types/>*
  <Schema/>*
  <Principals/>*
  <Policies/>*
</Interchange>
```

Generally, each of these components may appear more than once in any order. References need not be declared before they are used so long as they are declared somewhere in the document. The `<Schema>` sections define the vocabulary used in the policies. These include declarations for trust statements and actions. They may refer to standard XML Schema types, or custom types defined in the `<Types>` sections. The `<Policies>` sections contain definitions of policies defined in terms of the entities declared in `<Schema>` sections plus other entities imported from other FPI documents through `<Import>` components. The `<Principals>` sections collect frequently referenced principals and assign shorthand identifiers for them to be used in other parts of the document.

4.3.2 The top-level container

All FPI documents have a single root element `<Interchange>`. It includes a mandatory attribute `@targetNamespace`[2], which has type `xsd:anyURI`. This attribute introduces a namespace under which all the entities (including all trust statements, actions and policies) defined in this document will be scoped. An example is,

```
<Interchange xmlns="urn://opera.cl.cam.ac.uk/fidelis/FPI/04112001"
             xmlns:ns1="urn://opera.cl.cam.ac.uk/demo/test1"
             targetNamespace="urn://opera.cl.cam.ac.uk/demo/test1">
...
</Interchange>
```

The namespace identifier serves the purpose of version number, i.e. it is expected to be stable with the entity definitions and should change only if the accompanying definitions change. The attribute `xmlns` defines the default namespace of the elements in the document. The `xmlns:ns1` attribute binds the given namespace with a *prefix*, `ns1`. These are the standard mechanisms employed in XML Namespace [165].

4.3.3 Schema definitions

Schema definition sections contain definitions of trust statements and actions. A `<Schema>` element may have one or more `<Statement>` and/or `<Action>` elements, which define the entity structure. An example `<Statement>` definition is,

```
<Statement name="user">
  <Parameter name="UserID" type="xsd:string" />
  <Comment>
  The subject is a recognized system user, with the user name {UserID}.
  </Comment>
</Statement>
```

The corresponding definition in FPL is,

```
user (string UserID)
```

There may be zero or more occurrences of `<Parameter>` elements. A `<Parameter>` has a mandatory `@name` attribute, which gives the formal name of the parameter. Its type is given in a `@type` attribute, which may refer to a standard XML Schema type, types defined in `<Types>` sections, or from imported documents. The `@type` attribute may be omitted, in which case, the type definition must be given directly in its children using XML Schema `<complexType>` or `<simpleType>` fragments, e.g.

```
<Parameter name="UserID">
  <complexType>
  ...
  </complexType>
</Parameter>
```

`<Comment>` elements contain free-form text, intended to provide human-readable descriptions. They document the purpose of the entity and may also describe legal implications or guarantees. Actions are defined identically, except the element `<Action>` is used instead of `<Statement>`.

[2]As with the typical usage in XML standards, names prefixed with an “at” sign (@) denote attributes.

4.3.4 Principal declarations

Recall that principals are identified as public keys. As public keys are essentially long strings of numbers, they convey little meaning to humans, and additionally they can be inconvenient to work with. The primary intention of the `<Principals>` sections is to improve the readability of principal identifiers by assigning them with human-readable identifiers.

The informal syntax for the `<Principals>` element is given below ('+' means one or more occurrences and '?' means zero or one occurrence),

```
<Principals>
 (<Principal principalID="id" valueType="URI">
    <!-- content model depending on @valueType -->
  </Principal>)*
 (<Group principalID="id" valueType="URI">
   (<Principal principalRef="ref"? valueType="URI"?>
      <!-- content model depending on @valueType or ../@valueType -->
    </Principal>)+
  </Group>)*
 (<Threshold principalID="id" valueType="URI" threshold="integer">
    <!-- same as Group -->
    ...
  </Threshold>)*
</Principals>
```

A `<Principals>` element contains a number of `<Principal>`, `<Group>` and `<Threshold>` elements. A `<Principal>` element specifies a name in the `@principalID` attribute and contains a public key. A public key is given in the format indicated by the mandatory attribute `@valueType`. Currently the only supported format uses the XML Signature standard [166], with the identifier,

```
urn://opera.cl.cam.ac.uk/fidelis/FPI/04112001#xmldsig
```

Under this value type, the content of `<Principal>` contains an XML Signature `<ds:KeyInfo>` element.[3] The `<ds:KeyInfo>` element is a container for a wide variety of key information, ranging from plain DSA, RSA public keys, Base64-encoded PGP, SPKI certificates, to an XML representation of X.509 certificates. The design of FPI leverages and integrates with this work to provide a standards-compliant approach to specify public keys.

An example of a principal declaration section, which binds the identifier 'Alice' to a key, is provided below,

```
<Principals>
  <Principal principalID="Alice"
    valueType="urn://opera.cl.cam.ac.uk/fidelis/FPI/04112001#xmldsig">
    <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:RSAKeyValue>
        <ds:Modulus>xA7SEU+e0yQH5MICpzCCArm...</ds:Modulus>
        <ds:Exponent>AQAB</ds:Exponent>
      </ds:RSAKeyValue>
    </ds:KeyInfo>
  </Principal>
</Principals>
```

[3] Note that in this section, the prefix `ds` references the XML Signature namespace, `http://www.w3.org/2000/09/xmldsig#`

`<Group>` and `<Threshold>` elements assign a name to a group or a threshold principal respectively. They share an identical structure, which contains a list of `<Principal>` elements. This `<Principal>` element is similar but different from the `<Principal>` element directly inside the `<Principals>` element. It includes an optional attribute `@principalRef`, which refers to a named principal. It is however not permitted to define a name inside the enclosing scope, therefore the use of `@principalID` is prohibited. A `<Threshold>` element has a mandatory attribute `@threshold`, which gives the threshold value.

4.3.5 Policy specification

Policy specification is given in `<Policies>` elements. Each `<Policies>` element contains one or more `<TrustPolicy>` and `<ActionPolicy>` elements. These two elements correspond to trust policy and action policy specifications in the Fidelis Policy Language respectively. Consider an example, with the following trust statements defined,

```
user (string UserID)
admin (string UserID)
```

where the holder of a `user()` trust instance is a recognized system user, with the local user identifier as the parameter `UserID`, and similarly for `admin()`. Suppose the user ‘`pb`’ and ‘`maj`’ are the local system administrators. A trust policy can be written as,

```
user(a): self -> b
where a == ‘pb’ || a == ‘maj’
asserts admin(a): self -> b
```

The policy states that if a user is asserted to be either ‘`pb`’ or ‘`maj`’, a new trust instance `admin()` may then be issued for them. The equivalent trust policy in the interchange representation is as follows (assuming trust statements `user` and `admin` have already been defined),

```
<Policies>
  <TrustPolicy>
    <TrustUse name="ns1:user">
      <Parameter name="UserID" variableID="a" />
      <Truster self="true"/>
      <Subject variableID="b" />
    </TrustUse>
    <Where xsi:type="xsd:string"
     langType="urn://opera.cl.cam.ac.uk/fidelis/FPI/04112001#xpath2">
    $a = ‘pb’ or $a = ‘maj’
    </Where>
    <Asserts name="ns1:admin">
      <Parameter name="UserID" variableRef="a" />
      <Truster self="true" />
      <Subject variableRef="b" />
    </Asserts>
  </TrustPolicy>
</Policies>
```

A `<TrustUse>` corresponds to a *trust use* syntax component described in Section 3.10. It requires an attribute `@name` of type QName (an XML Schema type for namespace qualified names), which refers to the definition of a trust statement in the namespace. In this example, the namespace prefix `ns1` expands to a full namespace identifier. A `<TrustUse>` may contain one or more `<Parameter>` elements, a `<Truster>` and a `<Subject>` element. The purpose of `<Parameter>` elements is to bind a parameter to a variable placeholder. A `<Parameter>` element contains two mandatory attributes

@name and @variableID. The @name attribute identifies a parameter of the trust statement as declared, and @variableID assigns an identifier for a variable placeholder, which must be unique within the scope of the policy. It may be omitted if a parameter is not used in a policy, which has a similar effect of creating a unique but unreferenced parameter placeholder.

Both <Truster> and <Subject> share the same syntax. Only <Truster> will be referred to for brevity of exposition, but unless otherwise stated, the same description applies to <Subject>. The syntax is (where '|' means a choice),

```
<Truster principalRef="id"? variableID="id"? self="bool"?>
  (<Principal .../> |
   <Group .../> |
   <Threshold .../>)?
</Truster>
```

The attribute @variableID assigns a placeholder for the truster (and symmetrically, for the subject), and @self is a Boolean value, which is set **true** to refer to "this principal which has specified the policy". Note that the interpretation of @self is relative, therefore should be replaced with absolute principals when exporting policies to avoid ambiguity. The optional attribute @principalRef refers to a principal defined in the principal declaration sections. Principals may also be directly given in the content of <Truster> through elements <Principal>, <Group>, or <Threshold>. The syntax and semantics for these elements are identical to those in Section 4.3.4, except the use of attribute @principalID is not allowed.

Negative trust uses are given in <WithoutTrustUse> elements. They are similar to <TrustUse>, but in addition, may contain multiple <URL> elements, where each gives a repository where negative trust instances should be checked. The processing semantics requires checking with *any* of the listed repositories during policy evaluation. Please refer to Section 5.1.2 regarding distrust repositories.

The <Asserts> element corresponds to the **asserts** clause described in Section 3.5.5. It has a mandatory attribute @name which refers to the name of a trust statement. It has as children one or more <Parameter> elements, a <Truster> and a <Subject> element. There *must* exist one <Parameter> element for each parameter declared for the trust statement. It may either reference a parameter placeholder or specify a concrete value as its content. It has the syntax,

```
<Parameter name="string" variableRef="id"? environment="bool">
  <!-- optional content for a concrete, typed value -->
</Parameter>
```

where @variableRef references a placeholder bound previously through @variableID, and @environment indicates whether the value is supplied from the environment provided at policy evaluation time. This informs the policy processor that it is not an error if @variableRef references a non-existent variable. The <Truster> and <Subject> elements are similar to their counterparts in <TrustUse>, with the exception that @variableRef replaces @variableID.

A trust policy may also have zero or more <Grants> elements, which map to the **grants** clauses in Section 3.5.5. The syntax is a subset of <Asserts>, without <Truster> and <Subject> elements and where the @name attribute refers to an action.

The conditional and assignment clauses (**where** and **set**) are represented by <Where> and <Set> elements. These elements are designed to be extensible, i.e. the format of their contents depends on the extensibility identifier specified in the attribute @langType. The format currently supported is the predicate language in the XPath 2.0 standard [167]. XPath is an expression language for specifying paths in an XML document. In particular, it contains an XML Schema-aware predicate language for comparison, arithmetic, logical composition, and function calls. This format has the identifier,

urn://opera.cl.cam.ac.uk/fidelis/FPI/04112001#xpath2

An application may define an alternative format and declare it for another format identifier. Note that conditional and assignment clauses may often involve local computation (e.g. local database queries) and may reveal confidential security information. For these reasons, they can be explicitly hidden as follows,

```
<Where local="true" />
```

This informs the recipient of an FPI document that the policy is subject to local conditional (and/or assignment) computation in addition to the `<TrustUse>`s and `<WithoutTrustUse>`s.

Note that there is intentionally no representation for the validity clause. Recall that the main purpose of FPI is to enable interoperable policy distribution, while the determination of validity conditions for new trust instances is a deployment issue and should be at the discretion of the truster (i.e. where the policy is deployed). It is therefore meaningless to map the **valid** clause in the internal Fidelis Policy Language to the external policy representation of FPI.

The syntax for the specification of action policies is a subset of the trust policies. The `<ActionPolicy>` element is identical to `<TrustPolicy>` except for the absence of `<Asserts>` elements.

4.3.6 Linking with other policy documents

FPI is designed to enable distributed authoring of policies. Specifically it has a linking mechanism that helps the reuse of trust vocabularies across documents. This is especially useful when policies are authored in a top-down fashion, where a top-level authority may define a basic set of trust statements and actions, while leaving the specification of policies that use these definitions to subsidiaries. The linkage is achieved through `<Import>` elements. An example is given below:

```
<Import namespace="urn://opera.cl.cam.ac.uk/demo/test2"
        location="http://opera.cl.cam.ac.uk/demo/test2.fpi" />
```

The `@namespace` attribute indicates the namespace identifier defined in the FPI document which can be found at the URL given in `@location`. This element instructs the FPI processor to import the trust vocabulary defined in the referenced document (i.e. entities defined in the `<Schema>` and optionally the `<Types>` elements) into the current document. The imported namespace can be given a namespace prefix using the standard XML Namespace [165] mechanism. Qualified names, e.g. the `@name` attribute in `<TrustUse>` elements, can then be constructed using the namespace prefix to reference imported entities.

4.4 Credential representation

A Fidelis credential is essentially an extended public key certificate with a collection of typed attributes. There is a wide range of possible representations for Fidelis credentials, including SPKI certificates (which may include attributes as tagged values), X.509 version 3 certificates (which include application definable extensions), or indeed any version of X.509 certificates provided they are coupled with attribute certificates.

The web service implementation of Fidelis does not mandate any particular representation within a node. For example, a node may choose to use X.509 certificates to enable secure communication through SSL/TLS [164]. Instead, an XML-based representation is designed to enable interoperation between heterogeneous nodes. The format is known as the Fidelis Interoperable Credential (FIC) format.

4.4.1 Basic structure

Recall from Chapter 3 that a trust instance consists of an instantiated trust statement, a truster and a subject, a validity condition and a signature. In the FIC representation, a trust instance is an XML document fragment, whose top element identifies a namespace-scoped trust statement. It

contains three mandatory child elements: `<Truster>`, `<Subject>`, and `<Valid>`, plus an element for each parameter. It also contains at least one `<ds:Signature>` element, where the `<ds:Signature>` element refers to the container element of XML Signature [166].

Consider the following schema (adopted from Page 75),

```
<Statement name="user">
  <Parameter name="UserID" type="xsd:string" />
</Statement>
```

assuming its namespace (i.e. the value of `@targetNamespace` in its containing `<Interchange>` element) is,

```
urn://opera.cl.cam.ac.uk/demo/test1
```

An example trust instance would be (with some parts abbreviated):

```
<ns1:user xmlns:ns1="urn://opera.cl.cam.ac.uk/demo/test1">
  <UserID>wtmy2</UserID>
  <Truster>...</Truster>
  <Subject>...</Subject>
  <Valid type="status" ...>...</Valid>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">...</Signature>
</ns1:user>
```

The top-level element (`<ns1:user>`) is in the same namespace as that where the declaration of `user` belongs. Its parameter is given in the `<UserID>` element whose value is of the type for the corresponding parameter as declared (namely, `xsd:string`). Other elements provide information implied by their names, and are discussed in the rest of this section.

4.4.2 Truster and subject

The `<Truster>` and `<Subject>` elements contain principal identifiers for the truster and subject respectively. Both elements share the same syntax, given informally below:

```
<Truster>
  <Principal valueType="URI">
    <!-- @valueType elements -->
  </Principal>+
</Truster>
```

A `<Principal>` element must be associated with a `@valueType` attribute, whose value determines its content. There is currently one value type identifier, consistent with the description in Section 4.3.4,

```
urn://opera.cl.cam.ac.uk/fidelis/FPI/04112001#xmldsig
```

With this value type, the content of the `<Principal>` element contains a `<ds:KeyInfo>` element from XML Signature [166]. The set of `<Principal>` elements essentially provide a principal set. For trusters, this gives the set of principals who have signed the trust instance; for subjects, this indicates the set of principals for whom this trust instance is relevant. This may be used to satisfy group or threshold principals in policies, or may be used by a single principal in the set of subjects, depending how a request is made (and signed).

Note especially that there is no representation for threshold principals. The concept of threshold principals is for policy specification. The truster and subject set of a trust instance enable the enforcement of threshold-based policies. For example, suppose a policy states that at least three of the five management board members must approve a management decision. To satisfy this policy, it is sufficient to present a trust instance representing a management decision signed by a set of three management members.

Method	URI identifier
Timed CRL	`urn://opera.cl.cam.ac.uk/fidelis/04112001/tCRL`
Asynchronous timed CRL	`urn://opera.cl.cam.ac.uk/fidelis/04112001/tCRL-async`
Timed renewal	`urn://opera.cl.cam.ac.uk/fidelis/04112001/tRenewal`
Asynchronous timed renewal	`urn://opera.cl.cam.ac.uk/fidelis/04112001/tRenewal-async`
Status check	`urn://opera.cl.cam.ac.uk/fidelis/04112001/status`
Asynchronous status delivery	`urn://opera.cl.cam.ac.uk/fidelis/04112001/status-async`

Figure 4.7: `@method` URI identifiers for online validity schemes.

4.4.3 Validity condition

The validity condition of a trust instance is given in the `<Valid>` element. It supports both online and offline methods as described in Section 3.5.4. The type of validity method is specified through the mandatory `@type` attribute, which can be any of `offline`, `CRL`, `renewal` or `status`. For the offline method, the content of the `<Valid>` element is a pair of child elements, `<Start>` and `<End>`, whose values have the type `xsd:dateTime` from the XML Schema. For the online methods, the `<Valid>` element has a `@method` attribute and contains multiple `<URL>` elements. The `<URL>` elements specify locations from which the validity information may be obtained. The `@method` attribute indicates the interface method supported at those locations. An example online validity condition is provided below:

```
<Valid type="status"
       method="urn://opera.cl.cam.ac.uk/fidelis/04112001/status">
  <URL>http://opera.cl.cam.ac.uk/fidelis/status/</URL>
</Valid>
```

Six `@method` identifiers have been defined, shown in Figure 4.7. For a timed credential revocation list, an identifier is given to indicate the specified location that publishes an XML-based tCRL document, with hashed trust instances. Another identifier is given for the asynchronous version for obtaining this tCRL document. Two similar identifiers are allocated for timed renewal. For status query, identifiers are given for a SOAP-based synchronous query method and an asynchronous status delivery method. The example above indicates that the location, given in the `<URL>` element, supports the SOAP-based synchronous status query.

4.4.4 Signature

Signing of trust instances takes advantage of the XML Signature specification [166]. Trust instances use *enveloped* signatures in the terminology of XML Signature, which means a signature is enclosed within the signed document. XML Signature supports two algorithms for canonicalizing documents, with comments and without comments. Canonicalization is the process of stripping unneeded characters (e.g. whitespaces) from an XML document (or its fragment) resulting in a canonical representation. The signature in a trust instance is required to canonicalize without comments.

XML Signature supports a variety of *transformation algorithms*. A transformation is the process of deriving a set of elements from the canonicalized XML document for signing and verification purposes. The signature in a trust instance is required to use the *enveloped signature transform* (with the URI `http://www.w3.org/2000/09/xmldsig#enveloped-signature`), which essentially ignores the signature blocks when computing the set of elements for signing.

An example signature block for a trust instance is presented below (with long identifiers truncated):

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm="http://www.w3.org/..." />
    <SignatureMethod Algorithm="http://www.w3.org/..." />
    <Reference URI="">
      <Transforms>
        <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#..." />
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>k6kamhjv1tf0g...</DigestValue>
    </Reference>
  </SignedInfo>
  <KeyInfo><RetrievalMethod URI="#truster1"/></KeyInfo>
  <SignatureValue>MC0dbCFv3gkVrtt=...</ds:SignatureValue>
</Signature>
```

The signature may include a `<KeyInfo>` element that refers to the verification key, especially when there are multiple trusters. The `<KeyInfo>` element should contain the `<RetrievalMethod>` element with the attribute `@URI` which references the identifier for a truster using the XML referencing mechanism. In the above example, the signature points to the public key with the identifier `truster1`.

4.5 Summary

The emergence of web services represents the next step towards an open, global, and ubiquitous distributed computing platform. This chapter has presented a comprehensive design that implements Fidelis in the web service environment.

The web service environment consists of a collection of inter-communicating nodes, where each implements a number of *interfaces*. Five interfaces, collectively known as the Fidelis interfaces which facilitate trust management have been described. The *conveyance interface* allows trust instances to be exchanged between principals. The *trust inference interface* is the core of the architecture, which encapsulates the evaluation of policies and answers queries made against these policies. The *credential management interface* allows the management of trust instances, including collection, storage and retrieval. It is designed specifically to offload these tasks from small mobile devices, where resources are limited. The *policy interrogation interface* is designed to facilitate communication between strangers, so that unknown policies can be discovered through a query-based process. The *trust agent interface*, on the other hand, is designed to automate communication between strangers.

In our design, two types of information need to be exchanged in an interoperable manner: policies and trust instances. For consistency with web service technologies, XML representations have been designed for both types of data. Fidelis Policy Interchange (FPI) is essentially an XML version of the Fidelis Policy Language. It however extends FPL in several respects: namespace support, a type system (using XML/Schema), and a standards-based specification for principal identifiers. Trust instances are represented in the Fidelis Interoperable Credential (FIC) format, which leverages the XML Signature standard to provide integrity guarantees.

In the next chapter, we will focus on an algorithm that implements the semantics of the Fidelis policy inference process, and also describe an experimental framework enabling trust negotiation among strangers.

5 Inference and Trust Negotiation

At the heart of the Fidelis framework is the policy inference algorithm. The formal foundation underpinning the Fidelis policy language is first-order logic, which implies that the evaluation algorithms of Fidelis policies equate approximately to inferences in Prolog. However, some differences exist preventing standard Prolog inference algorithms (notably, the unification algorithms) to be directly applied to the Fidelis policy evaluation. These differences will be briefly described in this chapter. Section 3.5.8 presented the inference semantics from a conceptual point of view, stating the abstract goals and rules for processing a query. The first half of this chapter, Section 5.1, describes a concrete algorithm implementing the semantics described there.

One focus of the design of Fidelis and indeed the web services implementation described in the previous chapter is to cater for the open nature of the web where strangers may encounter each other and wish to communicate. Section 5.2 presents a trust negotiation framework whereby previously unknown parties may incrementally learn about each other, by dynamically discovering policies and selectively disclosing sensitive trust instances.

5.1 Policy inference

As previously mentioned, while the Fidelis Policy Language is based on first-order logic and its evaluation semantics are similar to the Prolog inferencing, there are two main differences between them significant enough to prevent the use of the standard Prolog unification algorithms for policy evaluation. First, negative trust in Fidelis policies needs special treatment that may require operational support. In the standard Prolog, because of the complexity implications, a restricted version of negation called *negation-by-failure* is typically employed. Such semantics are incompatible with Fidelis policies, as absence of trust in Fidelis merely means insufficient knowledge, as discussed in Section 3.5.3.

Second, Fidelis polices may contain parameters whose values are bound explicitly in **set** clauses, instead of using parameter matching rules. These parameters are similar to free variables in Prolog, and Prolog typically binds variables using unification, disallowing explicit “manual intervention” of variable bindings during the inference process. Evaluation of Fidelis policies requires more flexible handling of the way variables are bound.

To allow us to concentrate on the core algorithm, we have addressed the operational issues in separate subsections. In particular, the management of distrust repositories and the construction of validity conditions are not covered by our algorithm description. It is sufficient at the algorithm level to assume that a distrust repository is a database of trust instances of negative assertions.

5.1.1 Inference algorithm

On the interface level, an inference algorithm takes a set of trust instances, an environment and a query template (which embodies the query) as input, and outputs a Boolean result, and additionally, depending on the type of queries, one or more completed trust templates (on trust establishment,

vertical, horizontal and complete coverage queries), or action templates (on action decision or action coverage). Recall that a completed template is a template with all placeholders bound to instance values.

The inference semantics described in Section 3.5.8 can be divided into two phases.

- *Phase 1*: Policies that help in deriving the result need to be selected from the set of available policies.
- *Phase 2*: A subset of the input trust instances must be determined to satisfy those policies.

This semantics can be implemented using an algorithm that recursively resolves the conditions for a policy, until all conditions have been met or some condition has failed. This algorithm simultaneously addresses both phases in its recursion and may be implemented using an evaluation stack. For the sake of simplicity and clarity, the construction and tracking of validity conditions are discussed separately in Section 5.1.3.

We refer to the entity performing the actual inference as an *inference engine*. Suppose an *inference context* consists of an *evaluation stack* and an environment. Elements of the evaluation stack are called *evaluation contexts*. For each evaluation context, we write "[*state – info*]", where *state* may be one of LOOKUP, UNIFY, EVAL, NEG, SUCCESS and FAIL, and *info* gives state-specific information. The algorithm works by popping the top element from the evaluation stack and executing its operation. New evaluation contexts may be pushed back to the stack and the stack may be spawned if necessary, in which case, the process will be performed on both the original stack and the spawned one. The algorithm terminates on an evaluation stack when there is no more evaluation context left, or when the context of the state FAIL is reached.

We will first need to define the meaning of *matching* for a trust template to a trust instance, or to a trust policy.

Definition *Suppose* $tt = t(p_1, ..., p_n) : p_{truster} \rightarrow p_{subject}$ *is a trust template, where* t *is a name,* p_j *for* $1 \leq j \leq n$, $p_{truster}$ *and* $p_{subject}$ *are either variable placeholders or values. We say* tt *matches:*

- *a trust instance* $ti = i(val_1, ..., val_n) : val_{truster} \rightarrow val_{subject}$, *where* i *is a name,* val_j *for* $1 \leq j \leq n$, $val_{truster}$ *and* $val_{subject}$ *are values, if and only if:*
  - t *is equal to* i, *and*
  - *for any* $1 \leq k \leq n$ *or* $k = truster$ *or* $k = subject$ *and* p_k *is a value, then* val_k *is equal to* p_k.
- *a trust policy* tp *if and only if* tt *matches* tt_0 *where* tt_0 *is the trust template in the* ***asserts*** *clause of* tp.

Similarly, we define matching for an action template with action instances or action policies.

Definition *Suppose an action template* $at = a(p_1, ..., p_n)$, *where* p_j *for* $1 \leq j \leq n$ *are either variable placeholders or values. We say* at *matches:*

- *an action instance* $ai = i(val_1, ..., val_n)$ *where* val_j *for* $1 \leq j \leq n$ *are values, if and only if:*
  - a *is equal to* i, *and*
  - val_k *is equal to* p_k *for any* $1 \leq k \leq n$ *such that* p_k *is a value.*
- *an action policy* ap *if and only if* at *matches some* at_l, *where* at_l *is one of the action templates in the* ***grants*** *clause of* ap.

Let the set of all policies be T, the input set of trust instances be I, the current inference context be IC. We denote the evaluation stack of IC as $IC.S$ and its environment as $IC.E$. An environment consists of a set of variable bindings, denoted as $var = value$. The operation and context-specific information on evaluation contexts are described below:

[**LOOKUP** $- qm$] where qm is query template, i.e. either a trust or an action template. If qm is a trust template matched by some $i \in I$, then for each i, spawn the current inference context. Let the new inference context be IC'. Push [UNIFY – $i \Leftrightarrow qm$] onto $IC'.S$. For clarity of explanation, we avoid spawning on the *first* matching instance in this algorithm but instead work directly on the original IC.

Otherwise, if qm is a trust template without a matching $i \in I$, select a subset, T', of T such that it contains all matching trust policies, or if qm is an action template, all matching action policies. For each trust or action policy pl in T', spawn the current inference context (except for the first one, as above). Let the new inference context be IC'. Suppose pl is a trust policy,

$$pl = (\{t_1, ..., t_l\}, \{d_1, ..., d_m\}, cond, assign, t, \{a_1, ..., a_n\})$$

or if pl is an action policy,

$$pl = (\{t_1, ..., t_l\}, \{d_1, ..., d_m\}, \{a_1, ..., a_n\})$$

where t_j and d_j represent trust uses in the prerequisite and the **without** clauses respectively, *cond* and *assign* represent the expression in the **where** and **set** clauses, t is the trust template in the **assert** clause, and a_j is each action template in the **grants** clause. First, relabel each variable occurrence in pl so that conflicts with bindings in $IC'.E$ do not arise. Then merge the per-policy environment E_p, i.e.

$$IC'.E = IC'.E \cup E_p$$

On $IC'.S$, perform these in sequence,

- Push [SUCCESS – $t, \{a_1, ..., a_n\}$] if pl is a trust policy, or
- Push [SUCCESS – $\{a_1, ..., a_n\}$] if pl is an action policy.
- Push [EVAL – *assign*] if *assign* exits;
- Push [EVAL – *cond*] if *cond* exits;
- Push [NEG – d_j] for all d_j where $1 \leq j \leq m$;
- Push [LOOKUP – t_j] for all t_j where $1 \leq j \leq l$;

[**UNIFY** $- ti \Leftrightarrow tt$] where ti is a trust instance and tt is a trust template. Let $ti = i(val_1, ..., val_n) : val_{truster} \rightarrow val_{subject}$ and $tt = t(p_1, ..., p_n) : p_{truster} \rightarrow p_{subject}$, where i and t are names, val_j is a value, and p_j is either a value or a variable placeholder, $val_{truster}$ and $val_{subject}$ are principal values, and $p_{truster}$ and $p_{subject}$ may either be principal values or variable placeholders. Unification succeeds if and only if,

- For any $1 \leq j \leq n$, if p_j is a variable, it is either bound to val_j in $IC.E$ or unbound,
- For any $1 \leq j \leq n$, if p_j is a value, it is equal to val_j,
- If $p_{truster}$ is a principal value, $val_{truster}$ must be equal to $p_{truster}$; symmetrically for $p_{subject}$ and $val_{subject}$,
- If $p_{truster}$ is a variable placeholder and $p_{truster}$ is bound in $IC.E$, it must be bound to $val_{truster}$; symmetrically for $p_{subject}$ and $val_{subject}$, and
- Validation of ti must succeed.

If any of the above fails, push [FAIL] to $IC.S$. For any variable p_k unbound in $IC.E$, where $1 \leq k \leq n$, add $p_k = val_k$ to $IC.E$. Note that due to the design of the algorithm, action templates will never exist in a UNIFY evaluation context.

[**EVAL** – *expr*] If *expr* is a conditional expression, evaluate *expr* using bindings in $IC.E$ and push [FAIL] to $IC.S$ if the evaluation yields **false**, or do nothing on **true**.

If *expr* is an assignment expression, update bindings in $IC.E$.

[**NEG** – tm] where tm is a trust template. Let $tm = t(var_1, ..., var_n) : p_{truster} \rightarrow p_{subject}$ where t is a name, var_j is a variable placeholder, $p_{truster}$ and $p_{subject}$ may either be a variable placeholder or a principal value. Construct a trust template

$$tm' = t(val_1, ..., val_n) : val_{truster} \rightarrow val_{subject}$$

where for all $1 \leq j \leq n$, there exists $var_j = val_j$ in $IC.E$, and if $p_{truster}$ is a variable, there exists $p_{truster} = val_{truster}$ in $IC.E$, or if $p_{truster}$ is a principal value, $val_{truster}$ is equal to $p_{truster}$, and vice versa for $p_{subject}$ and $val_{subject}$. Then query for tm' in any of the associated distrust repositories and push [FAIL] if and only if the query result is positive.

[**SUCCESS** – $tm, \{at_1, ..., at_n\}$] where tm is a trust template and at_j for $1 \leq j \leq n$ is an action template. Note that tm will not exist for a SUCCESS context generated from action policies.

If $IC.S$ is empty, this means the final answer is reached. A new trust instance could then be constructed based on tm and a_j. If $IC.S$ is not empty, this indicates an intermediate result is reached. The inference engine may choose to construct a new trust instance or do other processing.

[**FAIL**] Abort the inference, causing an exception to be thrown and the inference context IC to be destroyed.

At the end of an inference run, because of the spawning operation, there may exist multiple inference contexts. Each inference context may provide a result, therefore there may be multiple results. These results constitute the answer for coverage type of queries.

Example

In this example, we use capital letters A, B, C and D to represent principals, and t_i for $1 \leq i \leq 5$ are trust statements. Lowercase letters are used as variables. For a description of the syntax, please refer to Section 3.5.5. Suppose A defines two trust policies:

$t_1(a) : \textit{self} \rightarrow p$ **asserts** $t_2(b) : \textit{self} \rightarrow p$ **set** $b := a + 20$ (P1)
$t_2(a) : \textit{self} \rightarrow p, t_3(b) : \textit{self} \rightarrow p$ **without** $t_4(a, b) : \textit{self} \rightarrow p$ (P2)
asserts $t_5(b) : \textit{self} \rightarrow p$ **where** $a > b$

Suppose a horizontal coverage query for t_5 is requested with the following trust template,

$$t_5(a) : A \rightarrow b$$

and with a set of trust instances:

$$I = \{t_1(10) : A \rightarrow B,\ t_1(20) : A \rightarrow C,\ t_3(20) : A \rightarrow B,\ t_3(40) : A \rightarrow D\}$$

A trace of the inference run is explained here. Inference contexts are enclosed in boxes, and bold typeface is used to highlight newly created evaluation contexts.

To begin the query evaluation, a new inference context, IC_0, is constructed to bootstrap. The evaluation stack is initialized with a LOOKUP context. The environment is initialized with the special variable *self* bound to the A plus other unbound variables in the query template, namely a and b.

1. $IC_0.S =$ [**LOOKUP** – $t_5(a) : A \rightarrow b$]
   $IC_0.E = \{\textit{self} = A, a =?, b =?\}$

   Because there exists no $i \in I$ that matches $t_5(a) : A \rightarrow b$, the set of matching trust policies is selected, $\{P2\}$. Because there is only one policy in the set, the original inference context is used instead of spawning. Occurrences of variable a and b in $P2$ are first relabelled to avoid conflicts with the environment. $P2$ effectively becomes,

$t_2(a') : self \rightarrow p, t_3(b') : self \rightarrow p$ **without** $t_4(a', b') : self \rightarrow p$
asserts $t_5(b') : self \rightarrow p$ **where** $a' > b'$

Note that *self* is not relabelled since it is a special global variable.

On $IC_0.S$, evaluation contexts representing $P2$ are pushed in. The bindings in the environment $IC_0.E$ are updated to reflect the introduction of new variables, e.g. a in $t_5(a)$ is bound to b'.

2. $IC_0.S$ = [**LOOKUP** – $t_2(a') : self \rightarrow p$]; [**LOOKUP** – $t_3(b') : self \rightarrow p$]; [**NEG** – $t_4(a', b') : self \rightarrow p$]; [**EVAL** – $a' > b'$]; [**SUCCESS** – $t_5(b') : self \rightarrow p$, {}]
$IC_0.E = \{self = A, a = b' =?, b = p =?, a' =?\}$

Similar to the above step, because no $i \in I$ matches t_2, a matching set of trust policies is selected, in this case $\{P1\}$. After relabelling, the policy becomes

$t_1(a'') : self \rightarrow p'$ **asserts** $t_2(b'') : self \rightarrow p'$ **sets** $b'' := a'' + 20$

Again, there is no need to spawn new inference context as the set has only one policy. Evaluation contexts representing $P1$ are therefore pushed onto $IC_0.S$ and environment $IC_0.E$ updated.

3. $IC_0.S$ = **[LOOKUP** – $t_1(a'') : self \rightarrow p'$ **]; [EVAL** – $b'' := a'' + 20$ **]; [SUCCESS** – $t_2(b'') : self \rightarrow p'$]; [LOOKUP – $t_3(b') : self \rightarrow p$]; [NEG – $t_4(a', b') : self \rightarrow p$]; [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : self \rightarrow p$, {}]
$IC_0.E = \{self = A, a = b' =?, b = p = p' =?, a' = b'' =?, a'' =?\}$

The processing of the LOOKUP evaluation context causes a copy of the inference context to be created since there are two possible matches for t_1 in I, $t_1(10) : A \rightarrow B$ and $t_1(20) : A \rightarrow C$. Let the new inference context be IC_1.

4. $IC_0.S$ = [**UNIFY** – $t_1(10) : A \rightarrow B \Leftrightarrow t_1(a'') : self \rightarrow p'$]; [EVAL – $b'' := a'' + 20$]; [SUCCESS – $t_2(b'') : self \rightarrow p'$]; [LOOKUP – $t_3(b') : self \rightarrow p$]; [NEG – $t_4(a', b') : self \rightarrow p$]; [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : self \rightarrow p$, {}]
$IC_0.E = \{self = A, a = b' =?, b = p = p' =?, a' = b'' =?, a'' =?\}$

Processing the UNIFY evaluation context updates the environment $IC_0.E$. Specifically, a'' is bound to 10 and p' is bound to B. For clarity, IC_1 is provided below. However for the rest of the discussion, we will only show IC_0, while the steps for IC_1 follow similarly.

$IC_1.S$ = [**UNIFY** – $t_1(20) : A \rightarrow C \Leftrightarrow t_1(a'') : self \rightarrow p'$]; [EVAL – $b'' := a'' + 20$]; [SUCCESS – $t_2(b'') : self \rightarrow p'$]; [LOOKUP – $t_3(b') : self \rightarrow p$]; [NEG – $t_4(a', b') : self \rightarrow p$]; [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : self \rightarrow p$, {}]
$IC_1.E = \{self = A, a = b' =?, b = p = p' =?, a' = b'' =?, a'' =?\}$

5. $IC_0.S$ = [EVAL – $b'' := a'' + 20$]; [SUCCESS – $t_2(b'') : self \rightarrow p'$]; [LOOKUP – $t_3(b') : self \rightarrow p$]; [NEG – $t_4(a', b') : self \rightarrow p$]; [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : self \rightarrow p$, {}]
$IC_0.E = \{self = A, a = b' =?, b = p = p' = B, a' = b'' =?, a'' = 10\}$

Note that the bindings for a'' and p' is updated as a result of processing the UNIFY context in the last step. The new top evaluation context is an assignment expression, which causes b'' to be bound to 30.

6. $IC_0.S$ = [SUCCESS – $t_2(b'') : self \rightarrow p'$]; [LOOKUP – $t_3(b') : self \rightarrow p$]; [NEG – $t_4(a', b') : self \rightarrow p$]; [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : self \rightarrow p$, {}]
$IC_0.E = \{self = A, a = b' =?, b = p = p' = B, a' = b'' = 30, a'' = 10\}$

Here we arrive at an intermediate result, $t_2(30) : A \rightarrow B$. Depending on the inference engine, it may create a trust instance accordingly, or simply record this fact for auditing purposes and continue the evaluation.

7. $IC_0.S$ = [LOOKUP – $t_3(b') : \mathit{self} \to p$]; [NEG – $t_4(a', b') : \mathit{self} \to p$]; [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : \mathit{self} \to p$, {}]
   $IC_0.E = \{\mathit{self} = A, a = b' = ?, b = p = p' = B, a' = b'' = 30, a'' = 10\}$

   Processing the LOOKUP context spawns one inference context, IC_2, since both $t_3(20) : A \to B$ and $t_3(40) : A \to D$ in I match $t_3(b') : \mathit{self} \to p$.

8. $IC_0.S$ = [**UNIFY** – $t_3(20) : A \to B \Leftrightarrow t_3(b') : \mathit{self} \to p$]; [NEG – $t_4(a', b') : \mathit{self} \to p$]; [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : \mathit{self} \to p$, {}]
   $IC_0.E = \{\mathit{self} = A, a = b' = ?, b = p = p' = B, a' = b'' = 30, a'' = 10\}$

   Processing the UNIFY context requires unifying the values for b' and p, whose corresponding values are 20 and B respectively. Since b' is unbound in $IC_0.E$, the unification succeeds. p is bound to B in $IC_0.E$, which agrees with its corresponding value, therefore also succeeds. After processing this evaluation context, $IC_0.E$ is updated with $b' = 20$.

   $IC_2.S$ = [**UNIFY** – $t_3(40) : A \to D \Leftrightarrow t_3(b') : \mathit{self} \to p$]; [NEG – $t_4(a', b') : \mathit{self} \to p$]; [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : \mathit{self} \to p$, {}]
   $IC_2.E = \{\mathit{self} = A, a = b' = ?, b = p = p' = B, a' = b'' = 30, a'' = 10\}$

   The processing is identical to that above; however, as p is bound to B in $IC_2.E$ while the corresponding value for p is D, the unification fails. This causes a [FAIL] context to be pushed to $IC_2.S$, which subsequently leads to the abortion of the evaluation and destruction of IC_2.

9. $IC_0.S$ = [NEG – $t_4(a', b') : \mathit{self} \to p$]; [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : \mathit{self} \to p$, {}]
   $IC_0.E = \{\mathit{self} = A, a = b' = 20, b = p = p' = B, a' = b'' = 30, a'' = 10\}$

   To process the NEG context, the inference engine first constructs a trust template by replacing variables with their bound values from the environment, resulting in $t_4(30, 20) : A \to B$. It then queries any one of the given distrust repositories. If a positive result is returned, it pushes [FAIL] onto the evaluation stack. For this discussion, we assume a negative result is returned.

10. $IC_0.S$ = [EVAL – $a' > b'$]; [SUCCESS – $t_5(b') : \mathit{self} \to p$, {}]
    $IC_0.E = \{\mathit{self} = A, a = b' = 20, b = p = p' = B, a' = b'' = 30, a'' = 10\}$

    Processing the EVAL context involves evaluating the expression $a' > b'$ with regard to the environment $IC_0.E$, which returns **true**.

11. $IC_0.S$ = [SUCCESS – $t_5(b') : \mathit{self} \to p$, {}]
    $IC_0.E = \{\mathit{self} = A, a = b' = 20, b = p = p' = B, a' = b'' = 30, a'' = 10\}$

    This reaches the final result for the query, which gives a trust template (with all parameters, truster and subject filled in), $t_5(20) : A \to B$. The inference engine may then create a trust instance by signing the template and return to the requester.

The runtime analysis of this algorithm will be discussed later in Section 5.1.4. We shall first discuss some operational issues with implementing this algorithm.

5.1.2 Managing distrust repositories

A policy may be associated with a list of distrust repositories. The algorithm described in the previous section works on the assumption that the set of distrust repositories must maintain a *consistent* view of distrust information. This is because it assumes it to be sufficient to check with *any* of the associated distrust repositories when processing a NEG evaluation context. While this requirement can be relaxed by changing the behaviour for the NEG evaluation context, this design cleanly separates operational issues from algorithmic ones, thus simplifying the inference algorithm.

The problem of maintaining consistency among distrust repositories is a standard problem of implementing replicated services. For distrust repositories, because of the security implication, it is crucial to enforce *strong consistency* among repositories. Strong consistency means that the publication

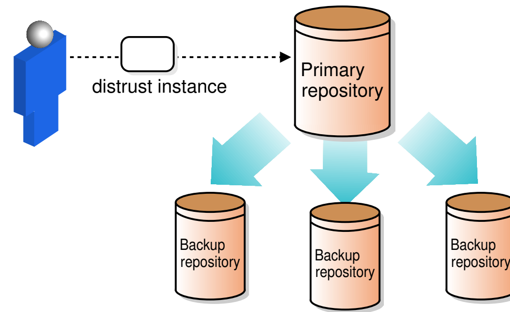


Figure 5.1: Passive replication scheme

of a trust instance to a repository will only be available if *all* repositories acknowledge its existence. The replication problem for distrust repositories is however simpler than, for example, replicated file services because the only update operation is *append*, which does not cause conflicts between replicas.

One approach for strong consistency is to use quorum assembly with an atomic commit protocol such as the two-phase commit protocol. Such a protocol ensures all replicas in the quorum reach the same decision for an operation, either commit or abort. The decision is then propagated to other replicas. Nevertheless, because of the sensitivity of distrust information, it may be undesirable to abort the publication of trust instances.

Another approach is to apply the *passive replication* scheme, as illustrated in Figure 5.1. One of the distrust repositories is elected to act as the primary replica. A principal publishes a trust instance through the primary repository, which in turn forwards the update to backup repositories. This scheme keeps strong consistency by ensuring the update operation (i.e. the publication of trust instances) can only be done through the primary replica, while all replicas (both primary and backup) may handle query requests. In order to deal with network partitioning, the reachability between every backup repository and the primary repository must be monitored. If the primary repository is not reachable either because of machine crash or network partitioning, a backup repository should stop or downgrade its service, e.g. returning an “unknown” status when a queried trust instance is not in its repository instead of returning a definite “no entry” response. Under such circumstances, a new primary repository may be elected using some election protocol to resume normal services.

5.1.3 Tracking validity

The algorithm described in Section 5.1.1 deliberately separates the construction of validity conditions from the inference process. Recall that the validity condition for a new trust instance may either be explicitly specified in the **valid** clause of a trust policy, or implicitly derived from its prerequsite trust instances. It is therefore necessary to address both implicit and explicit construction of validity conditions in the algorithm.

To add support for explicit validity conditions, the SUCCESS evaluation context needs to be augmented to include the validity condition as specified in the **valid** clause. Recall that a SUCCESS context is pushed into the evaluation stack when a trust policy is decomposed. Processing a SUCCESS context should then use the contained validity information to create the validity condition for the new trust instance.

For implicit validity conditions, there must be a mechanism for computing the weakest validity condition among prerequisite trust instances as described in Section 3.5.8. Toward this goal, the inference context IC is augmented with a validity condition, denoted $IC.V$. $IC.V$ serves as temporary storage for the weakest validity condition discovered so far. For descriptive convenience, we assume it may be one of **always**, **period**, or **status**. $IC.V$ is initialized to **always** when IC is created. During the inference process, on a [UNIFY $- ti \Leftrightarrow tt$] context, after a successful unification between ti and tt,

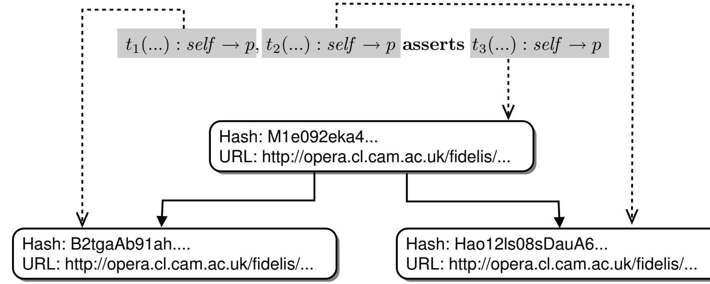


Figure 5.2: Validity dependency tree

the validity condition of ti must be merged with $IC.V$, according to the rules defined in Section 3.5.8. The new validity condition is stored back at $IC.V$. When the inference steps are completed, the $IC.V$ will contain the weakest validity condition.

A separate issue regardless of whether the new validity condition is derived implicitly or explicitly is the dependency between online validity conditions. Suppose the following trust policy is defined:

$$t_1(...) : \mathit{self} \rightarrow p, t_2(...) : \mathit{self} \rightarrow p \textbf{ asserts } t_3(...) : \mathit{self} \rightarrow p$$

Also suppose that appropriate trust instances for t_1 and t_2, whose validity conditions are both online status checks (i.e. **status**), are given to obtain an instance of t_3. The validity condition for t_3 will be **status** by the validity computation rules. However, the online validity of the t_3 instance should be subject to the validity of t_1 and t_2 instances. If either the t_1 or t_2 instance invalidates, so should the t_3 instance.

One approach to track this dependency is to use a *validity dependency tree*. Each node of the tree contains sufficient information to query the validity of a trust instance, e.g. a location and a hashed value of the trust instance. A parent-child link represents a validity dependency, i.e. the validity of the parent depends on the validity of all its children nodes. Figure 5.2 illustrates a validity dependency tree. Solid lines represent links between nodes, while dashed lines represent the correspondence between parts in a trust policy and nodes. A node contains two pieces of information: a hash value for a trust instance and a URL location where the validity status can be obtained.

The construction of a validity dependency tree can be integrated with the inference algorithm. We first augment the inference context IC with a tree construction stack $IC.VS$. The elements of the stack are the nodes waiting to be added to a tree. Initially, $IC.VS$ is empty. On encountering a [UNIFY – $ti \Leftrightarrow tt$] context with the validity condition of ti being **status**, suppose the unification between ti and tt succeeds, a node representing ti's validity condition is created and pushed into $IC.VS$. On a [SUCCESS – $tm, \{at_1, ..., at_n\}$] context, the inference engine shall create a new validity condition with all the nodes in $IC.VS$ popped out and made as its children. The new validity condition is then pushed into $IC.VS$. This process continues until the inference algorithm terminates. On a success, $IC.VS$ will be left with one element, which is to be the root node of a dependency tree. The children nodes will already be properly constructed.

When the status query of a trust instance is requested, its dependency tree should be consulted, traversing each node and collecting status information. When the traversal is completed, the status of the root node can then be determined.

5.1.4 Runtime analysis

Before the runtime of the algorithm can be analyzed, it is essential to discuss the termination property of the algorithm. The termination of the algorithm is subject to the input policies. The input policies must be *cycle-free*, otherwise the algorithm may be non-terminating on certain queries. A *cyclic*

policy is one whose result trust instance is either directly or indirectly one of its own prerequisite trust instances. A straightforward, although somewhat artificial, example would be:

$$t_1(a) : \mathbf{self} \to p \ \mathbf{asserts} \ t_1(a) : \mathbf{self} \to p$$

where t_1 is the name of a trust statement, and a and p are variable placeholders. A more complicated and realistic example would be:

$$t_1(a) : \mathbf{self} \to p \ \mathbf{asserts} \ t_2(a) : \mathbf{self} \to p \quad \text{(P3)}$$
$$t_2(a) : \mathbf{self} \to p,\ t_3(a) : \mathbf{self} \to p \ \mathbf{asserts} \ t_1(a) : \mathbf{self} \to p \quad \text{(P4)}$$

where t_1, t_2, and t_3 are names for trust statements, and a and p are variable placeholders. These two policies form a cycle since in order to assert a t_2 instance, a t_1 instance is needed. However, to obtain that t_1 instance, the same t_2 instance would be needed. This hence creates a cycle, and the queries to obtain either t_1 or t_2 instances will lead the algorithm into an endless recursion.

The algorithm described in this chapter is in fact a recursive version of a depth-first search over the set of input policies. Let the input set of policies be P and the input set of trust instances be I. Let N be the total number of *terms* of P, where a term includes a *trust use*, *distrust use* or *trust template* in a trust policy (please refer to syntax 3.21 on Page 48 for details). Finally, let M be the maximum number of parameters in any trust instance.

The runtime of the algorithm depends mainly on the processing of [LOOKUP] contexts. We shall therefore first consider the maximum possible number of [LOOKUP] contexts in any query run. A query starts with one [LOOKUP] context. The processing of a [LOOKUP] context might generate either a [UNIFY] context or more [LOOKUP] contexts. Suppose on the processing of the ith [LOOKUP] context, the number of new [LOOKUP] contexts is x_i. The total number of [LOOKUP] contexts is hence $\sum^i x_i$. However, recall that a new [LOOKUP] context is generated for each *trust use* in a policy. Since the maximum number of terms is N, and P is acyclic, the total number of [LOOKUP] contexts for any query is thus:

$$\sum^i x_i = O(N) \tag{5.1}$$

We shall now analyze the cost for processing each [LOOKUP] context. When processing a [LOOKUP] context, the algorithm first searches for a trust instance matching the query template of the context in I. Provided trust instances in I are indexed by hash values of their digests, the search may be done in constant time, $O(1)$. If a match is not found, the algorithm attempts to search for matching policies in P. Similarly, policies in P may be indexed by the names of *trust templates* or *action templates* for trust and action policies respectively. The search would therefore be done in $O(1)$ time. The total time for this case is:

$$O(1) + O(1) = O(1) \tag{5.2}$$

If a matching trust instance is found in I, a [UNIFY] context is created. Processing a [UNIFY] context involves comparing all parameter values in a trust instance and a trust template. This operation is linear, and as the maximum number of parameters is M, it would cost $O(M)$ time. The total runtime for this case is therefore (where $O(1)$ is the time to search a match in I):

$$O(1) + O(M) = O(M) \tag{5.3}$$

Comparing with Equation 5.2, we know that the worst runtime for processing any [LOOKUP] context is $O(M)$. Combining with Equation 5.1, we shall conclude that the runtime for processing any query would cost:

$$O(N) \times O(M) = O(MN) \tag{5.4}$$

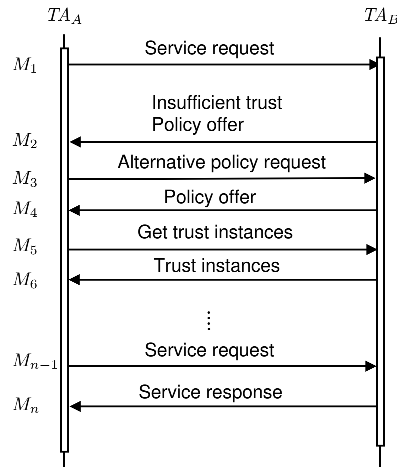


Figure 5.3: A trust negotiation session

5.2 Trust negotiation

As discussed previously in Section 4.2.6, trust negotiation is an approach to facilitate communication between unfamiliar principals. The aim is to enable two strangers to gradually gain trust in each other, and subsequently, provide services, vouch for the stranger, etc. This section first describes a trust negotiation framework, followed by a description on *meta-policies*, which drive the behaviours of trust negotiation sessions. There are two aims of this negotiation framework: first, as an experiment validating Fidelis; second, as an experimental framework for future research on negotiation protocols.

5.2.1 Trust negotiation overview

Trust negotiation is embodied as *negotiated requests*, where a request may be a service request or a trust establishment request. It is carried out between a pair of principals or trust agents acting on behalf of some principals. We write TA_A for the trust agent representing principal A and TA_B for B. Suppose A attempts to establish a trust relationship with a stranger B, i.e. obtaining a trust instance from B regarding A, it makes this request to TA_A, which in turn communicates with TA_B and successively exchanges trust instances until the trust request satisfies the requirements set by B. The session of exchanging trust instances is called a *trust negotiation session*. A trust negotiation session is governed by a negotiation protocol, which defines the messages and their flow. The behaviour of trust agents TA_A and TA_B may be defined through a set of *meta-policies*. A meta-policy defines the conditions when a trust instance or policy can be disclosed. It is discussed fully in Section 5.2.3.

A sample protocol session is illustrated in Figure 5.3. TA_A initiates a request for gaining a trust instance in message M_1. TA_B examines the relevant trust policy of B and decides that more trust instances are needed to satisfy the request. It thus replies with an *insufficient trust* exception and offers some policies to TA_A (M_2). TA_A may decide to choose other policies than those offered. In this case, it sends an *alternative policy* request (M_3) back to TA_B. In order to gain more trust from TA_B, it may attach some trust instances along with the request. With more knowledge about A, TA_B replies with another, perhaps less stringent set of policies in M_4. If a policy offered by TA_B requires TA_A to reveal some sensitive trust instance, TA_A may wish to first ensure TA_B has the rights to see it. This could be achieved by explicitly asking for some trust instance of B, as represented by M_5 and M_6. This conversation continues until TA_A and TA_B establish a mutually agreed policy or abort. In the prior case, TA_A then finally re-sends the request with reference to the negotiated policy (M_{n-1}). TA_B then passes this request on to B, which generates a response (M_n).

The detail of the protocol is described in the next section.

5.2.2 Trust negotiation protocol

The protocol is based on a sequence of request/response messages. There are nine types of message:

- general request (GR),
- general request result (GRR),
- insufficient trust exception (ITX),
- policy offer (PO),
- alternative policy request (ALT),
- credential request (CR),
- credential disclosure (CD),
- general abort request (GA).
- general abort acknowledge (GAA).

A general request message represents either a trust establishment request or an action request. It consists of the real request, an optional policy (or its reference), and an optional set of trust instances. The trust instance set provides assertions about the requester for satisfying the trust requirement of the request. The policy sent along with a GR message is offered and signed by the responder. This policy may be a general policy that applies to the same type of request, but is more likely to be a tailor-made policy for the specific request concerned. A general request result is a computation result of the request. For example, on a trust establishment request, the result may be a new trust instance or a refusal; a *read* action on a file may return the content of the file.

An insufficient trust exception indicates that the presented set of trust instances does not meet the trust requirement for a request. It may be accompanied by a policy offer message, which contains a set of signed policies granting the requested operation. An alternative policy request is sent when the requester wishes to find out other policies that may be offered by a responder, e.g. if it does not wish to comply with any of the offered policies. The message may contain a set of trust instances, which provides more information about the requester, in the hope of it thus convincing the responder to allow it to obtain less strict policies.

A credential request message is similar to the *getTrustInstance* method in the conveyance interface described in Section 4.2.2. It consists of a trust template, and effectively does a simple lookup for a trust instance. Unlike *getTrustInstance*, it is associated with a negotiation session and the result may be influenced by beliefs learned within the session. A credential disclosure message contains a set of trust instances that satisfy the trust template of a CR message.

A general abort message is for forcibly terminating a negotiation session. Session states are normally deleted at the end of the session, either due to a normal termination or abort. Typically garbage collection should also be invoked at a fixed interval to prevent stale session states left by crashed or uncooperative negotiating entities.

The protocol action is described as a finite state machine. The states correspond to the sending and receiving of each message, e.g. GR SENT, ALT RCVD, CR SENT, etc. There are two special states, START and DONE as the initial and the completion states respectively. Figure 5.4 shows the finite state machine, without showing the states for general abort messages to simplify the figure. A general abort may be initiated by either side, at any RCVD state, i.e. some message has been received. The lines connecting states represent occurrences of some event, solid lines for events on the requester TA, dashed lines for the responder TA. For the protocol action, we shall describe only the requester side, i.e. the solid lines. The one for the responder side mirrors the requester side.

A requester initiates a negotiation session by sending a general request message with an empty policy and a possibly empty set of trust instances since the requester does not know the requirements for authorizing the request. In the GR SENT state, it may receive the result for the request, which causes the session to end successfully in the DONE state. It may receive a CR message, which means

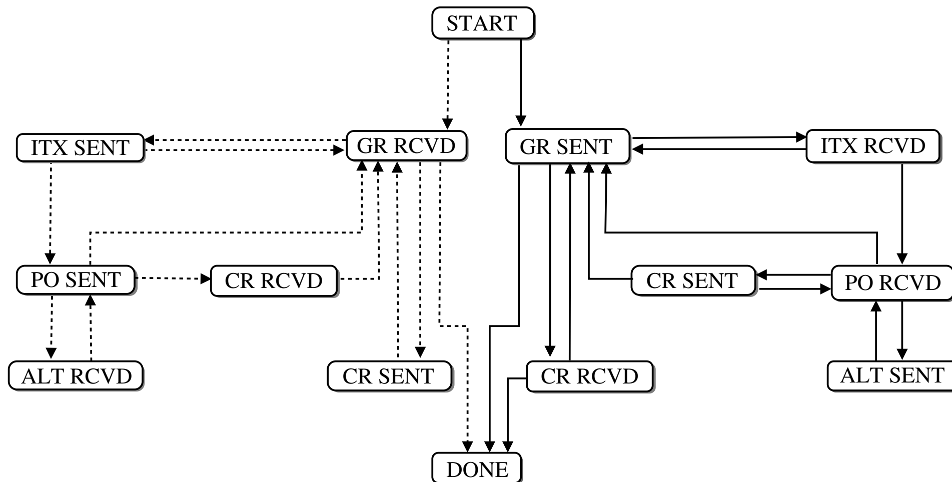


Figure 5.4: State diagram for the negotiation protocol. The solid lines indicate the path for the requester TA, while the dashed lines indicate the path for the responder TA.

the responder attempts to directly obtain trust information to grant the request. If the requester has the requested trust instance, the requester may decide if it wishes to disclose it. If disclosed, the responder should return the result for the request; if not disclosed, it moves back to the GR SENT state so that other options may be attempted. In GR SENT, the requester may receive an insufficient trust exception, taking into the ITX RCVD state. In this state, the requester may retry the request with more trust instances, thus back into the GR SENT state. The responder may send a policy offer message along the exception, resulting in the PO RCVD state.

In the PO RCVD state, the requester examines the offered policies, chooses appropriate trust instances and retries the request (PO RCVD→GR SENT). If the offered policies require a disclosure of some sensitive trust instances, the requester may wish to first check the responder's trust information. It may hence send a CR message (PO RCVD→CR SENT). It may decide to renegotiate for another set of policies, by providing more information about itself. It sends an ALT message and the responder may return a new set of policies or nothing (i.e. an empty policy set). In the CR SENT state, the requester either receives a trust instance matching its query or nothing. In the former case, it discloses the sensitive trust instances concerned and retries the request (CR SENT→GR SENT). In the latter case, the requester needs to re-examine the offered policies, and returns to the PO RCVD state.

As previously mentioned, a general abort may take place after any message is received. Specifically, on the requester side, it may wish to abort a session at state ITX RCVD, PO RCVD, and CR RCVD, and it should be prepared for a GA message at state GR SENT, ALT SENT and CR SENT.

5.2.3 Meta policies

During a trust negotiation session, trust agents on both sides need to determine which trust instances and/or policies can and should be disclosed. These are particularly important if they contain sensitive information, i.e. their disclosure may hamper security or cause privacy invasion. An impractical approach is to have human involvement in a negotiation session so that human principals on either side will review and decide what information is to be given, according to some guideline or rules.

In an attempt to automate this decision process, a FPL profile for *meta-policies* is described. Meta policies serve two purposes:

- defining disclosure criteria for trust instances used in a general request message, an alternative policy request or in response to a credential request message.

Entity	Description
$disclose(item)$	An action representing the disclosure of *item*.
$negotiator() : self \rightarrow p$	Stating principal p is acted on behalf by the negotiating agent.
$agent() : self \rightarrow p$	Stating the identity of the negotiating trust agent as p.
$disclosed(item) : self \rightarrow self$	Stating that a protected item *item* has been disclosed at some point in this negotiation session.
$presented(item) : self \rightarrow self$	Stating that *item* has been presented by the negotiating party at some point during the current session.
$requested(action) : self \rightarrow self$	Stating that *action* is requested in the current negotiation session.

Figure 5.5: Vocabulary for the meta-policy profile

- defining criteria governing the conditions when policies might be offered.

They are based on a denial-by-default rule, i.e. if a trust instance or policy is not explicitly allowed to disclose in a context (i.e. to a principal, to an action request, etc), it is considered as confidential. The vocabulary for this profile is designed to support four types of disclosure policies: *designated principal disclosure*, *context-specific disclosure*, *trust-directed disclosure*, and *mutual exclusion*. The vocabulary is summarized in Figure 5.5. Trust instances are used to represent facts known within a session. Hence their trusters are always *self*, i.e. the trust agent itself. These may be constructed from the context, e.g. if a trust agent is representing another principal requesting a `check_balance` operation, this is represented as:

```
requested(check_balance(41245516)): self->self
```

They may also be facts learned from the negotiation process, e.g. if the negotiating party sends a `bank_branch` trust instance, this would be represented as:

```
presented(bank_branch("20-17-19"): p1 -> p2): self->self
```

In Figure 5.5, *items* are referred to as *protected items*, which include both confidential trust instances and policies. For trust instances, they are given as trust templates, as shown above. For policies, they are given as *policy identifiers*. A policy identifier is assigned to every protected policy, and may be grouped to form a *policy group*, which is also identified by a policy identifier. The *action* in Figure 5.5 represents an action template with some or all parameters filled in.

We shall present some examples, demonstrating the use of the vocabulary for the four types of disclosure policies. In the following examples, `T1`, `T2`, ... are used for trust statements, and `A1`, `A2`, ... are for actions. We shall ignore their parameters where they are irrelevant to the meta-policies. We will continue the prior convention of using lowercase letters for variables and readable names for principal identifiers.

Designated principal. Protected items can be made available to only some designated principals. Indeed, this is a common constraint in real life, e.g. a trust instance containing personal banking details, e.g. account number, branch number, etc. may be restricted for use at the bank itself. An example meta-policy would be:

```
negotiator(): self->p
```

```
where p = Alice
grants disclose(T1(...))
```

This specifies that trust instance `T1` matching the template in the **grant** clause may be disclosed only when negotiating with Alice.

Context-specific disclosure. In addition to constraining to specific principals, it is often desirable to express constraints at the granularity of specific requests. For example, one may allow the task of purchasing goods at a well-known online store to use only trust instances for credit rating and address proof. The trust statement `requested` allows this type of policy to be specified, as follows,

```
requested(A1(...)): self->self
grants disclose(T2(...))
```

This states that the matching `T2` trust instances may only be disclosed if the request is the specific `A1` enclosed in the `requested` trust use. Note that a **where** clause may be included to constrain parameters in both `A1` and `T2`.

Trust-directed disclosure. The basis of trust negotiation is to exchange and gradually disclose trust instances based on those presented by the negotiating parties, with the aim to reach a sufficient level of trust on both sides for the requested operation. The trust statement `presented` is designed precisely to allow specification of this type of disclosure policies.

As an example, suppose a commercial service offers two classes of services: basic and premium. The access to these services is governed by the category of the customer. If a customer subscribes to the premium service, he or she will be identified by a `premium_user` trust instance. Suppose the service only intends to disclose the access policies for premium customers if it is negotiating with a premium customer, a meta-policy could be specified,

```
presented(premium_user(...): p1->p2): self->self
grants disclose("premium-policies")
```

The string `premium-policies` identifies the set of policies for premium customers. This meta-policy allows the identified policies to be disclosed in a policy offer message, provided the negotiating party proves that it possesses a `premium_user` trust instance.

Mutual exclusion. Mutual exclusion policies specify two or more trust instances should not be disclosed within the same session. This is potentially useful if these trust instances, when linked, would allow unnecessary or even sensitive information to be inferred. For example, suppose a principal only wishes to be identified as an employee of a company but does not wish to disclose her salary if she has disclosed the company she works for. Mutual exclusion may be specified by combining `disclosed` trust statements and the **without** clause. For example,

```
without disclosed(T3(...): p1->p2): self->self
grants disclose(T4(...))
```

states if an instance of `T3` has been disclosed, then disclosure of `T4` instances is prohibited. Note that the `disclosed` statement is instantiated for a trust instance and made known to the trust agent whenever the trust instance is sent to the negotiating party.

The above examples demonstrate relatively simple uses of the meta-policy vocabulary. Complex policies may be expressed by combining these trust statements. The key design of the meta-policy profile is the use of `presented` and `disclosed` trust statements, which effectively represent historic events within a negotiation session. Their inclusion adds a temporal dimension so that policies governing the interaction of protocol sessions can be specified.

5.2.4 Related work

Automated trust negotiation (ATN) has only recently attracted interest from the research community although the idea of mechanizing negotiation to reach common understanding is not new. Indeed, the de-facto Internet security protocol, SSL/TLS [164], is a prime example of credential-based negotiation. The SSL protocol however has an assumption, driven by its protocol design, which requires the server to disclose its credentials before learning anything about the client. The client may then be required to submit its own credentials in exchange. However, SSL does not provide a mechanism for the client to interrogate the server. (Note that such a mechanism does exist for the other direction, i.e. the server interrogating the client). SSL can be considered as an early attempt at a specific type of negotiation – identity authentication.

Winsborough et al. [168] describes a comprehensive trust negotiation framework with a similar goal set out in this section. They model a negotiation session as a sequence of *credential disclosures*, and each disclosure is guarded by a *credential access policy* (CAP). A CAP may be satisfied by the requester disclosing a set of other credentials, similar to the *trust-directed disclosure* described in the previous section. Their framework includes two negotiation strategies. In the *eager* strategy, as soon as a CAP is satisfied, the credential will be disclosed. In the *parsimonious* strategy, credentials with satisfied CAPs are only disclosed if they are needed to satisfy other CAPs. However, their negotiation strategies essentially hard-code the protocol behaviour whereas the Fidelis negotiation framework advocates a fully policy-driven approach – each principal may define its own meta-policies that control the protocol behaviour, which gives an increased flexibility.

In most prior work on ATN, the only entities that are assumed to contain sensitive information are the credentials. Seamons et al. [169] identifies that policies may also be sensitive in practice and proposes an extended trust negotiation framework which allows policies to also be subject to protection. They employ a similar mechanism to protect policies, and also support two strategies for controlling the disclosure of policies. The Fidelis negotiation framework is designed to provide a uniform treatment of both trust instances and policies – the policy-driven mechanism does not distinguish the types of protected entity. With the policy-driven protection of policies, the Fidelis negotiation framework is therefore more flexible than the hard-coded strategies in Seamons' ATN framework.

A recent work by Winsborough et al. [170] examines and partly addresses the inferential disclosure of credentials. Their observation is that most prior work on ATN is under the assumption that all parties are sane and honest, and will follow the negotiation strategy as specified. However, Winsborough noted that by observing the responses to certain types of request, a party may attempt to link and derive sensitive credentials held by the other party. For example, let `A`, `B`, `X`, and `Y` be trust statements. Suppose Alice has defined a trust relationship:

```
A(...): self -> p asserts B(...): self -> p
```

Now suppose an instance of `B` contains sensitive information, and its disclosure policy requires an instance of `X` to be first presented by the negotiating party. Further, suppose the disclosure policy of an instance of `A` requires presenting an instance of `Y`. Finally, suppose Bob who does not possess instances of `X` but does have an instance of `Y` may infer, with a high probability, that Alice does own an instance of `B` by finding out that she has `A`.

They propose a partial solution to the problem. The idea is that the response for querying `A` and `B` should be uniform, thus disabling the negotiating party to infer whether Alice has either `A` or `B`. They extended the notion of credential access policy (CAP) to *acknowledgement policy* (AP). Essentially an acknowledgement policy is an access policy for policies. For example, an AP may be associated with the trust relationship above, demanding the presentation of a valid `X` before disclosing `A`, `B` and the policy itself. While the Fidelis negotiation framework is not specifically designed to address the inference problem discussed here, its policy-driven disclosure protection of policies may achieve a similar effect for acknowledgement policies, i.e. by explicitly specifying the trust instances that must be known before a policy itself may be disclosed. Furthermore, the Fidelis negotiation framework allows the specification of relationships between policies, i.e. allowing the disclosure of policy A provided policy

B is not disclosed. To the best knowledge of the author, Winsborough's acknowledgement policy does not have provision for this type of linkage of policies.

5.3 Summary

At the centre of Fidelis is its policy inference algorithm. In the last chapter, an informal semantics was provided. This chapter has described an algorithm that implements these semantics. Additionally, the design and implementation issues for managing distrust information and tracking validity conditions have been discussed.

This chapter has also presented a trust negotiation framework that trust agents may implement to enable strangers to gradually gain trust in each other and subsequently perform trust-based requests. A trust negotiation protocol has been described. In addition, an FPL profile for specifying meta-policies that control the behaviour of automated negotiation sessions has been described. The primary innovation of this negotiation framework, in contrast with most other prior work, is its fully policy-driven approach. The most appealing feature of this approach is its increased flexibility and extensibility. The framework has been positioned as an experimental platform for future research on automated trust negotiation.

In the next chapter, we shall examine Fidelis operating in a number of real-world application scenarios.

6 Applications

In previous chapters, we have described fragments of several applications to illustrate the use of the policy language and motivate our design for web services. The focus of this chapter is to describe a number of case studies, providing detailed studies of the use of Fidelis in real-world applications. It begins with a study on implementing role-based access control (RBAC) using Fidelis in Section 6.1. Two prominent models of RBAC are examined, the OASIS RBAC [3, 5] model and the RBAC96 (and derivative) model [8], and Fidelis is shown to successfully model policies in both models. Section 6.2 describes Fidelis for the World Wide Web. In particular, it describes the integration of Fidelis with the popular Apache web server [171] and provides a number of examples demonstrating its use. In the last case study, we describe an electronic marketplace, consisting of multiple, independent parties. The primary goal is to illustrate the use of Fidelis in a decentralized, cooperating environment.

6.1 Role-based access control

As the research of role-based access control matures, we are beginning to observe a growing adoption of RBAC in operating systems, database management systems, and general applications. As described in Chapter 2 (Section 2.1.5), with the concept of *roles*, RBAC has clear advantages over the traditional DAC or MAC in its scalability, flexibility and manageability. For a system with a large number of users, RBAC simplifies security administration – the assignment of users to their appropriate roles, and privileges to these roles. As the number of roles will typically be significantly less than the number of users, this increases both scalability and manageability of the system. RBAC is also more flexible in the sense that both MAC and DAC may be simulated by properly configuring RBAC policies.

Fidelis may be used to model RBAC policies. The underlying idea is to consider roles as a property of a principal, and expressing the membership of a role with trust instances. In this section, we shall discuss the use of Fidelis to express two distinct RBAC models: the OASIS RBAC model [3, 5] and the RBAC96 derivatives [8, 40, 41].

6.1.1 OASIS role-based access control

The OASIS RBAC model builds on the basic concepts of RBAC, separating three types of base entities: users[1], roles and privileges. In addition, it introduces the notion of *appointments* and *environment predicates*. Chapter 2 includes a brief review of OASIS. Here we summarize the key features of its RBAC model.

- Parametrized roles and privileges. Roles may contain parameters to include attributes specific to a particular role member, e.g. the local user identifier. Parameters in a privilege enable the

[1] The term *user* is used interchangeably with *principal* in the OASIS model and in most RBAC models. In OASIS, a user refers to a *user session* [5].

specification of fine-grained authorization policies, e.g. a parameter for **read** might give the pathname to the requested file.

- Session-based roles. Every user works within some session, within which roles may be activated. Only privileges of active roles may be exercised within a session. Deactivation of roles is based on an automatic, chained revocation, which may be triggered by the termination of a session.
- Policy-based. The rules for role activation are specified in *activation rules*. Activation may be subject to three types of conditions: prerequisite roles, appointments and evaluations of environment predicates. Privileges are assigned to roles through *authorization rules*.
- Appointments as persistent credentials. In addition to session-based roles, appointments (which are similar to parametrized roles) are included for applications which require an extended lifetime beyond sessions for maintaining information about principals.

Rules in OASIS are written in the syntax of first-order logic. Consider an example from [5]. An activation rule

$$\textit{A_employed(username?)}, \textit{E_is_doctor(username, dept?)} \vdash \textit{R_doctor(username, dept)}$$

where *A_employed* is an appointment, *E_is_doctor* is an environment predicate and *R_doctor* is a role. In this thesis, we follow a naming convention for these OASIS entities, where the prefix A_, E_, R_, and P_ indicates an appointment, an environment predicate, a role or a privilege respectively. Parameters affixed with a question mark (?) are *out-parameters*, whereas parameters without a trailing question mark are *in-parameters*.

Upon evaluation, if an out-parameter exists in a role or appointment, it binds to the value of the corresponding parameter. If it exists in an environment predicate, the evaluation of the predicate must set its value upon completion. For an in-parameter, it obtains the binding from a previously bound out-parameter with the matching parameter symbol. For example, in the above rule, *username* is first bound to the corresponding parameter in an *A_employed* appointment instance. The value is then given as an input to the evaluation of *E_is_doctor*. The semantics of an activation rule is that every antecedent (i.e. conditions on the left hand side of the $\vdash$ symbol) must be satisfied for the consequent to be activated. Satisfaction is subject to correct bindings of parameters.

An authorization rule is in a similar form. For example,

$$\textit{R_treating_doctor(username?, pat_nhs_id?)} \vdash \textit{P_read_health_record(pat_nhs_id)}$$

where *R_treating_doctor* is a role and *P_read_health_record* is a privilege. The evaluation semantics are identical to activation rules. The meaning of this rule is therefore: the requester must prove he/she is a treating doctor of a patient within the current session for the request of reading the patient's health record to be granted.

Mapping into Fidelis policies. Fidelis is derived from the work on OASIS, with a degree of semantic similarity in their policy languages. Hence a near-perfect mapping from OASIS policies into the Fidelis counterparts is possible.

Semantically, OASIS users equate to Fidelis principals. Nevertheless users in OASIS are implicit. It is always assumed that at policy evaluation, the only user concerned will be the requester. As will be seen later, the explicit treatment of principals in Fidelis permits greater control and flexibility.

Appointments are mapped into trust statements. OASIS appointments are intended to express *task assignment* and *qualification*. For example, suppose Alice is employed at Hospital A with the employee identifier "aek322". She may be given an appointment instance by the human resource department of the hospital to testify her status as an employee:

A_employed ("aek322")

The corresponding trust instance would be:

```
A_employed("aek322") : HospitalA -> Alice
```

This trust instance means that the truster (i.e. the human resource department at Hospital A) believes that the subject (namely, Alice) is a legitimate employee of the hospital, with the employee identifier "aek322". Note that the trust statement approach explicitly states the issuer and the subject of an appointment instance, which may be used to aid policy specification.

Fidelis actions correspond directly to OASIS privileges. Recall that at specification, actions have explicit requesters, whereas in OASIS authorization rules, the requesters are implicit. An example will be provided later to illustrate this difference.

In the OASIS model, roles are always bound to sessions. There is no concept of "inactive roles" as in other RBAC models (see Section 6.1.2). Membership of a role may be expressed as a Fidelis trust statement, shown in the template form below:

as_⟨*role name*⟩ (s, ...)

where the eclipses correspond to the parameters of the role, and s represents a session identifier. An instance of this trust statement means that the subject is currently active in the role within session s. There are two approaches to provide the value for s within a trust or action policy: it may either be provided as a part of an input environment at evaluation time, or alternatively, it may be explicitly set through a **set** clause. In OASIS, names of roles and appointments exist in different namespaces, whereas as both are mapped into Fidelis trust statements, care therefore must be exercised to avoid conflict of names.

Environments in OASIS are designed for two purposes: (1) specifying constraints on parameters, and (2) assigning values to parameters. In Fidelis, these are achieved through **where** and **set** clauses. An OASIS environment predicate is therefore decomposed and mapped into expressions in **where** and **set** clauses.

Validity conditions for trust instances that represent appointment are often in the form of expiry periods, as appointments are intended to express long-lived facts. For facts independent of time limits, e.g. a degree one has earned, **permanent** validity conditions may be used. However, for roles, because of the session-based nature, online status would be appropriate as validity conditions for as_⟨*role name*⟩ instances. Such status would reflect the status of a session, i.e. set to **true** when a session is initiated, and **false** when a session terminates.

We will now consider several examples adopted and modified from [5] and demonstrate the specification of OASIS-style RBAC policies in Fidelis, using the mapping rules described previously. These examples share a common background of an electronic healthcare system.

Example 6.1 Suppose every employee in Hospital A is issued with an appointment instance *A_employee*, which has one parameter – the local username. A doctor may then use her appointment instance to activate the role *R_doctor*. Suppose the activation rule is:

A_employed(username?), E_is_doctor(username, dept?) ⊢ *R_doctor(username, dept)*

where *E_is_doctor* is an environment predicate that takes a *username* and returns **true** if *username* is a doctor and sets the doctor's department in *dept*. The same rule coded as a Fidelis trust policy would be:

```
A_employed(username) : self -> p                                    (P1)
asserts as_doctor(s, username, dept) : self -> p
where "do a database query to determine whether username is a doctor."
set dept =  "username's department from a database query"
        s = "new session identifier"
```

Note that **self** in the policy above binds to Hospital A's public key identifier when the policy is deployed. Besides the syntactical differences, in the OASIS rule, the user is hidden from the specification, while in the Fidelis counterpart, there is an explicit treatment of both the policy owner (in this case,

Hospital A) and the principal concerned. In this case, one is required to show an *A_employed* trust instance issued by the specific principal represented by **self**, and after evaluation, a new trust instance of *as_doctor* is explicitly bound to the same principal as the subject of the presented *employed* trust instance. A new session identifier will be generated and assigned to the variable s upon successful evaluation. This contrasts to the implicit session management in the OASIS counterpart.

Example 6.2 Suppose a doctor on duty in a clinic will be active in role *R_doctor_on_duty*. An on-duty doctor may be assigned to provide treatment to outpatients. This assignment takes place after an outpatient arrives and registers at the clinic. The registrar staff will then issue an appointment instance *A_patient_assigned* to the appropriate doctor. With this appointment instance, an on-duty doctor may then activate the *R_treating_doctor* role for the specific patient. In OASIS activation rules, this could be expressed as:

R_doctor_on_duty(username?, dept?), *A_patient_assigned (pat_nhs_id?)* $\vdash$
R_treating_doctor(username, pat_nhs_id)

where *pat_nhs_id* is the patient's unique NHS identifier, *username* and *dept* are respectively the doctor's local user identifier and his/her serving department. The equivalent policy in Fidelis would be:

```
as_doctor_on_duty(s, username, dept) : self -> p                    (P2)
A_patient_assigned(pat_nhs_id) : self -> p
asserts as_treating_doctor(s, username, pat_nhs_id) : self -> p
```

Note that the subjects for both *as_doctor_on_duty* and *patient_assigned* instances in the Fidelis formulation are explicitly required to match because of the variable matching semantics. The explicit treatment of principals also increases the flexibility. For example, as a principal may be a group principal, a *patient_assigned* instance may be issued to a group of doctors, allowing activation by any of the doctors in the group.

Example 6.3 The ultimate goal of RBAC is to enable authorization decisions. Following from the examples above, suppose one authorization rule for accessing a patient's health record is that the requester is one of the patient's treating doctors, i.e. is in an appropriate *R_treating_doctor* role. In OASIS authorization rules, this can be expressed as:

R_treating_doctor(username?, pat_nhs_id?) $\vdash$ *P_read_health_record(pat_nhs_id)*

where *P_read_health_record* is a parametrized privilege that takes an NHS identifier, *pat_nhs_id*. This rule states that in order to be granted with read access of a patient's record, a requester must be active in a *R_treating_doctor* role with the patient's NHS identifier as a parameter. The same rule can be coded as a Fidelis action policy:

```
as_treating_doctor(s, username, pat_nhs_id) : self -> p             (P3)
grants P_read_health_record(s, pat_nhs_id) : p
```

In this formulation, the requester for action *P_read_health_record* must match with the subject of the presented *as_treating_doctor* instance, namely, the doctor who is currently active in the role. Note that however, *P_read_health_record* takes an additional parameter, the session identifier, s, whose value is obtained through the session identifier contained in the *as_treating_doctor* instance.

6.1.2 RBAC96 and the NIST unified model

RBAC96 [8] is a family of four models, RBAC_0, RBAC_1, RBAC_2, and RBAC_3. RBAC_0 is the base model, defining users (U), roles (R) and privileges (P) and association between users and roles (user assignment, UA), and roles and privileges (privilege assignment, PA). It also defines the notion of *sessions* (S), where sessions contain active roles. RBAC_1 builds on RBAC_0 and adds the notion of *role hierarchy*. This will be discussed in more detail later. RBAC_2 adds to RBAC_0 with *constraints*.

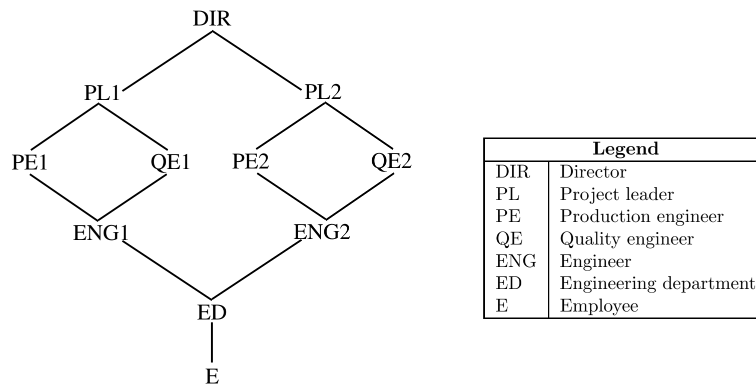


Figure 6.1: An example role hierarchy (adopted from [8]).

The most important constraint is the *separation of duty* constraints, also discussed later. $RBAC_3$ is a combined model of $RBAC_1$ and $RBAC_2$. A recent attempt to unify the diversity of RBAC models resulted in the *NIST unified RBAC framework* [40, 41]. The framework describes three levels of RBAC models: core RBAC, hierarchical RBAC and constrained RBAC. These are essentially $RBAC_0$, $RBAC_1$ and $RBAC_3$, with an extension for review functions. For example, user-role review returns the set of roles a user is assigned to, including those inherited; role-user review does the opposite, and role-privilege review returns the set of privileges a role is directly granted or inherited. The review functions are intended to help administrators inspect the configuration of RBAC policies.

Role hierarchy is a partial order on roles, based on the "seniority" relation. Figure 6.1 shows an example of a role hierarchy, adopted from [8]. In this figure, senior roles are shown above junior roles. There exists several interpretations for role hierarchies. The most common ones are *privilege inheritance* [8, 38, 41] and *activation hierarchy* [44]. In the privilege inheritance interpretation, a role inherits privileges assigned to all its junior roles, including transitive ones. For example, in Figure 6.1, role PL1 will have privileges granted to PE1, QE1, ENG1, ED, and E. In the activation interpretation, a user assigned to a role may activate any of its junior roles in a session, including transitive ones. So that if a user is assigned to PL1, she may activate PE1, QE1, ENG1, ED, or E in a session. Effectively, the user is *implicitly* assigned with those roles.

As reviewed in Chapter 2, separation of duty is a mechanism for decomposing a task into subtasks, and assigning them to different users for execution in order to increase security and protect integrity. For RBAC, several types of separation of duty constraints have been discussed in the literature [46, 31, 32]. The more commonly agreed concepts are: static separation of duty (SSD) and dynamic separation of duty (DSD). Before discussing these types of constraint, it is worth noting that the notion of sessions is designed specifically to enable the support for separation of duty constraints and promote the principle of least privilege. By activating a subset of all assigned roles, a user can obtain "just enough" privileges for the current task and avoid violating separation of duty constraints.

Static separation of duty defines a mutually exclusive set of roles that must not be *assigned* to the same user. This places constraints on the assignment of users to roles (i.e. the UA relation). Dynamic separation of duty works on a weaker basis, allowing mutually exclusive roles to be assigned to the same user, but preventing them from being simultaneously active in the same session. While SSD is simple, it is a stronger constraint than DSD and may thus be inflexible for practical use.

Expressing RBAC96-style models. Fidelis policies can be written to express the semantics of the RBAC96 and derivative models. A RBAC96 user equates to a Fidelis principal. Therefore U, the set of all users, becomes the set of all principals in the system. Privileges in RBAC96 are simple atoms, which can be modelled as parameter-less actions in Fidelis. P, the set of all privileges, therefore

User	Assigned roles
Bob	{ E, ENG1, PE1 }
Cathy	{ E, ENG1, QE1 }
Dave	{ E, ENG1, PE1, PL1 }
Eve	{ E, DIR }

Figure 6.2: Role memberships for users in the examples.

maps into the set of all actions. Roles in RBAC96 can be expressed as trust statements, so that a trust instance represents the subject's membership of a role. However, as RBAC96 distinguishes between active roles and assigned but inactive roles, a role needs to be represented by two trust statements:

`as_`⟨*role name*⟩`(s)`

and

`assigned_`⟨*role name*⟩

where ⟨*role name*⟩ is the name of a role and s is a session identifier. An instance of the `as_`⟨*role name*⟩ statement means that the subject is active as a member of the role ⟨*role name*⟩, while an instance of the `assigned_`⟨*role name*⟩ statement indicates that the subject is assigned with the role ⟨*role name*⟩ and may activate it for use in some session. A trust policy captures the role activation mechanism:

`assigned_`⟨*role name*⟩ `: self -> p asserts as_`⟨*role name*⟩`(s) : self -> p`

The binding value for s may be provided from an input environment at evaluation, or from a **set** clause. This is similar to the treatment in the previous section.

Assignment of users to roles and privileges to roles are defined as relations in RBAC96. User assignment is defined as $UA \subseteq U \times R$ and privilege assignment is $PA \subseteq P \times R$. In Fidelis, information in UA is mapped into trust policies, and members of PA are mapped into action policies. For example, suppose a partial user assignment relation for the role hierarchy in Figure 6.1 is as given in Figure 6.2. The same information may be expressed as trust policies:

```
asserts assigned_E : self -> 1-of { Bob, Cathy, Dave, Eve }        (P4)
asserts assigned_ENG1 : self -> 1-of { Bob, Cathy, Dave }          (P5)
asserts assigned_PE1 : self -> 1-of { Bob, Dave }                  (P6)
asserts assigned_QE1 : self -> 1-of { Cathy }                      (P7)
asserts assigned_PL1 : self -> 1-of { Dave }                       (P8)
asserts assigned_DIR : self -> 1-of { Eve }                        (P9)
```

Note that these policies produce trust instances with subjects as threshold principals. The semantics of threshold principals allows any threshold number (in this case, 1) of the members in the specified group to use the trust instance. An alternative approach would be setting subjects in additional **set** clauses, possibly through database queries with greater flexibility at the cost of verbosity. Privilege assignment (PA) may be expressed in a similar fashion, albeit using action policies instead.

Role hierarchies are defined as a partial order on R, $RH \subseteq R \times R$, written as $\geq$, e.g. if $r_1 \geq r_2$, then r_1 is directly senior to r_2. A role hierarchy may be expressed as a set of action policies or trust policies depending on which interpretation to use.

Privilege inheritance. With this interpretation, a role inherits privileges that all of its junior roles are assigned with, and junior roles include those transitively defined. Suppose role r is assigned with a privilege $pv \in P$. In Fidelis, this is:

```
as_r : self -> p grants pv : p
```

Then for each role $r_i \in R$ such that $r_i \geq r$, add an action policy:

```
as_ri : self -> p grants pv : p
```

As an example, suppose role QE1 in Figure 6.1 is assigned with a privilege P1. Under privilege inheritance, this assignment causes the following action policies to be introduced in an atomic step:

```
as_QE1 : self -> p grants P1 : p        (P10)
as_PL1 : self -> p grants P1 : p        (P11)
as_DIR : self -> p grants P1 : p        (P12)
```

Activation hierarchy. This interpretation of a role hierarchy enables a user to activate roles she is assigned with, plus additional roles junior to those assigned. For a role r, its activation would be specified as:

```
assigned_r : self -> p asserts as_r : self -> p
```

Activation of junior roles may be expressed as, for each $r_i \in R$ such that $r \geq r_i$,

```
assigned_r : self -> p asserts as_ri : self -> p
```

Based on this formulation, the sub-hierarchy rooted from QE1 of the hierarchy in Figure 6.1 may be expressed as the following trust policies:

```
assigned_QE1 : self -> p asserts as_QE1 : self -> p        (P13)
assigned_QE1 : self -> p asserts as_ENG1 : self -> p       (P14)
assigned_QE1 : self -> p asserts as_ED1 : self -> p        (P15)
```

Note that the Fidelis formulation for role hierarchies does not replace the seniority relation (namely *RH*). Instead, it serves as a complement to the information in *RH* and provides a semantics for the relation. In RBAC96 and derivatives, the interpretations are provided through textual definitions.

The basis of expressing separation of duty constraints is through the distrust mechanism (i.e. the **without** clause). Separation of duty constraints are specified in the RBAC96 family of models as sets of mutually exclusive roles. We shall consider static separation of duty (SSD) and dynamic separation of duty (DSD) separately.

Static separation of duty. SSD enforces mutual exclusion of user assignment to roles. That is if roles r_1 and r_2 are mutually exclusive, they cannot be both assigned to the same user at any time. An approach to express this constraint as employed in RBAC96 is as a set $SSD \subseteq 2^R$, where each member of *SSD* specifies a set of mutually exclusive roles.

To express SSD constraints in Fidelis, **without** clauses need to be added to every trust policy that represents user assignment for mutually exclusive roles. More specifically, for every $rs \in SSD$ and $r \in rs$, the user assignment policy for r would have the form:

```
without assigned_r1 : self -> p, ..., assigned_rn : self -> p
asserts assigned_r : self -> p
```

where $r_i \in (rs - \{r\})$ for $1 \leq i \leq n$, and $|rs - \{r\}| = n$. As an example, suppose role PE1 and QE1 in Figure 6.1 are mutually exclusive in *SSD*. The assignment policies P6 and P7 would thus become:

```
without assigned_QE1 : self -> p                           (P6')
asserts assigned_PE1 : self -> 1-of { Bob, Dave } as p
without assigned_PE1 : self -> p                           (P7')
asserts assigned_QE1 : self -> 1-of { Cathy } as p
```

The first policy ensures PE1 can be assigned to Bob or Dave if and only if they have not already been assigned with QE1. The second policy ensures that QE1 can only be assigned to Cathy if and only if she has not already been assigned with PE1. The combination of these two policies hence correctly implements SSD between PE1 and QE1.

Dynamic separation of duty. DSD enforces mutual exclusion on role activation. It is also captured as a set in RBAC96, namely, $DSD \subseteq 2^R$, whereby each member is a set of mutually exclusive roles. If two roles r_1 and r_2 are in the same set, they may be assigned to the same user but must not be activated simultaneously in the same session.

DSD constraints may be specified in Fidelis as **without** clauses in trust policies for role activation. For every $rs \in DSD$ and $r \in rs$, the activation policy for r would be:

```
assigned_r : self -> p
without as_r1(s) : self -> p, ..., as_rn(s) : self -> p
asserts as_r(s) : self -> p
```

where $r_i \in (rs - \{r\})$ for $1 \leq i \leq n$, and $|rs - \{r\}| = n$. Note that the variable for session identifiers (i.e. s) must match across trust statements for active roles to ensure a faithful modelling of the DSD semantics. Suppose PE1 and QE1 in the previous example are mutually exclusive in DSD (instead of SSD), the trust policies for activation of PE1 and QE1 will be:

```
assigned_PE1 : self -> p without as_QE1(s) : self -> p
asserts as_PE1(s) : self -> p
assigned_QE1 : self -> p without as_PE1(s) : self -> p
asserts as_QE1(s) : self -> p
```

Constrained by the **without** clause in the first policy, a principal assigned to PE1 may only activate it if and only if he/she is not already active in QE1 in the same session. The second policy works symmetrically and completes the mutual exclusion.

6.1.3 Discussion

As shown in this section, Fidelis may express a variety of role-based access policies. For OASIS policies, there exists a near-perfect mapping to Fidelis policies. The major difference in the policy specification between OASIS and Fidelis is that principals are implicit in the former whereas they are treated explicitly in the latter. In OASIS, it is assumed that when a policy is evaluated, there is only one principal in the context, i.e. the role owner and/or the requester. In Fidelis, a policy may be evaluated in a context where there exist multiple principals, as trusters and/or subjects.

Explicit treatment of principals in Fidelis provides additional power and flexibility. It allows the specification of, for example, threshold-based access control, which is difficult in OASIS. Moreover, it enables the specification of access policies for *proxied requests*, i.e. requests passed through a chain of intermediary entities. In this case, a request appears to the destination service as if initiated by the last-hop intermediary. Fidelis policies can be easily written to distinguish principals, whereas in OASIS, this is awkward at best.

Fidelis can also express the policies of RBAC96 and its derivatives. It allows precise modelling of role assignment, role activation, role hierarchy and separation of duty constraints. In general, the policy-based approach of Fidelis is more verbose than RBAC96's set-based specification, especially when used to express hierarchies. However, the policy-driven approach defines a clear semantics. For example, for role hierarchies, it clearly defines the intended interpretation, as privilege inheritance or activation hierarchy.

Because of the verbosity of the Fidelis approach for RBAC96 policies, it is conceived that a policy tool may be constructed so that role policies may be defined and manipulated graphically. Appropriate policies may then be generated automatically from the graphical specification. This may both ease policy management and reduce human error.

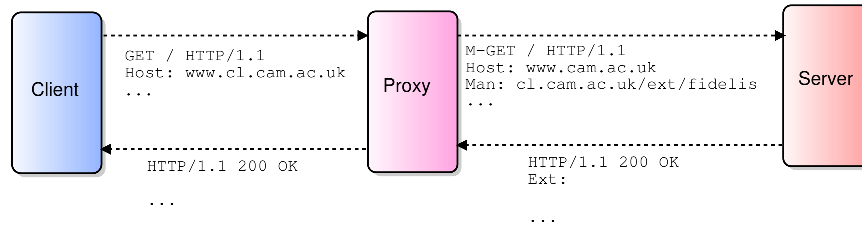


Figure 6.3: Proxy mechanism supporting Fidelis trust management.

6.2 Case study: Trust management in the World Wide Web

Since its inception in the early 90s, the World Wide Web (WWW) has rapidly established itself as the "killer application" of the Internet. However, authorization management for the WWW has traditionally been ad-hoc with lack of any uniform framework. The goal of this case study is to apply Fidelis trust management to the WWW, devising a platform on which new styles of collaborative applications may be built. The prominent types of application include:

- Collaborative content management systems. A web site may determine the access of its content (e.g. parental control, pay-for-content), or customize its content for the intended audience based on the trust instances issued by third parties.
- Single sign-on systems. Users often need to maintain independent identities for each site they use, e.g. different online shops, web space providers, pay-for-content sites, etc. Single sign-on systems aim to provide a portable identity across multiple sites. Existing solutions are usually based on some centralized database, whereas the trust management approach offers an attractive privacy-respecting alternative.

The main consideration for the design is that it should introduce minimal or no changes to the existing WWW architecture, and it should build on standards where possible. Moreover, where changes to the architecture cannot be avoided, they should only be made at the server end, not the client end. This is due to the fact that the WWW is a well-established and mature technology. Introducing architectural changes would severely limit the practical applicability and acceptance of the solution, or at the very least the speed of its adoption.

6.2.1 Architectural overview

The aim of this case study is to integrate Fidelis into the existing WWW architecture in a seamless fashion, thereby enabling trust instances to be used to assist the process of content authorization, generation and delivery from a web server. Towards this aim, it is essential to associate trust instances with HTTP requests (which underly the WWW). This implies the web client (usually a web browser) needs to include a set of trust instances of the user for requests it sends, and the web server needs to interpret and perform appropriate actions based on the submitted trust instances. Where necessary, trust negotiation may need to be initiated alongside the HTTP interactions.

In order to minimize the impact on the web client, our design makes use of the standard HTTP proxy mechanism. As described in the HTTP RFC [172], a proxy is an intermediary program that acts simultaneously as a server and a client and is intended to make requests on behalf of some clients. A proxy may transform requests from a client, in which case, it is called a *non-transparent* proxy. The proxy designed for Fidelis trust management is a non-transparent proxy, called the *Fidelis smart proxy*.

The overall architecture is shown in Figure 6.3. A client issues a standard HTTP request, which is serviced by a smart proxy which transforms the client request into an extended request, associated

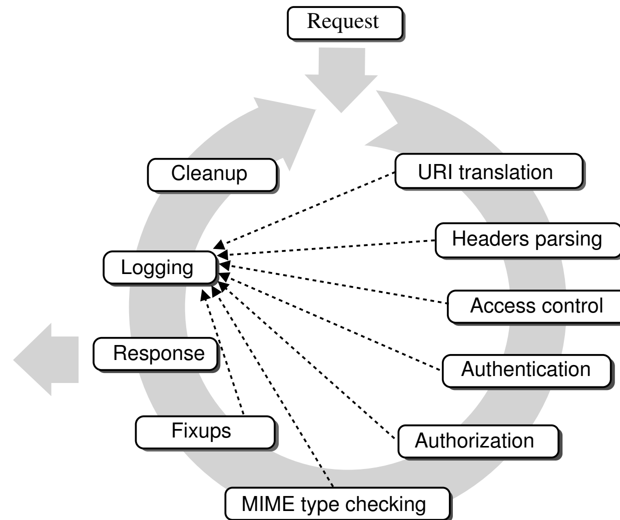


Figure 6.4: Request handling cycle in the Apache server (version 1.x)

with a set of trust instances in the FIC (Fidelis Interoperable Credential) format. The server then needs to interpret the request and its associated trust instances, and perform relevant actions such as making an authorization decision, customizing contents, etc. The extended request makes use of the HTTP Extension Framework [173], which allows custom extensions to be created for the HTTP protocol. For a response, the smart proxy simply relays the information returned by the server to the client.

This architecture does not require any change to the web client, and hence satisfies our basic design requirement. The smart proxy essentially acts as a trust agent as described in Section 4.2.6, and may initiate trust negotiation with the server. Note that the smart proxy is transparent to servers. The extended request produced by a smart proxy must appear as if it is initiated by the client itself. The requester hence must bind the requester identity, using the mechanism described in Section 4.2.7.

The server, however, needs to be extended to include Fidelis functionality. There are two main functions that a Fidelis-aware web server must handle. First, it must understand the HTTP extension that the Fidelis smart proxy uses, and second, it must be able to expose its mechanism for handling HTTP requests as Fidelis actions so that action policies may be written to control its behaviour. For this case study, it was decided to integrate Fidelis into the popular open-source Apache web server [171]. The next two sections describe the extension work for Apache version 1.3.

6.2.2 Request handling in Apache

The Apache web server features a modular framework, whereby modules may be dynamically loaded to enrich the server. Typical tasks performed by modules include transforming HTTP requests, invoking external CGI (Common Gateway Interface) programs that generate dynamic content, redirecting requests, and implementing custom authorization schemes. A module consists of a number of hooks (or *handlers*) that are invoked by Apache at appropriate times for two main purposes: to parse the configuration file and to modify the behaviour of request handling. We shall discuss the latter in more detail.

Apache breaks request handling into several stages as illustrated in Figure 6.4. When a request is received, it first translates the URI (Uniform Resource Identifier) in the request into a local filename where possible. It then parses the HTTP headers into a hash table and in addition, performs some processing against these headers.

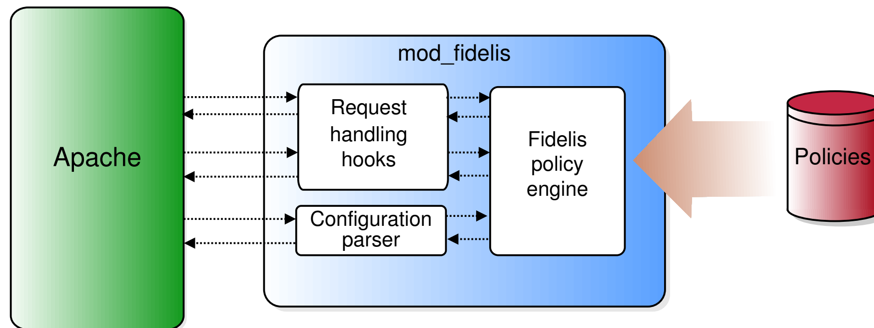


Figure 6.5: Architecture of `mod_fidelis`

The next three stages are related to determining whether access to a page should be granted. These three stages are named: *access control*, *authentication* and *authorization*. The terminology is somewhat confusing. Essentially the *access control* stage refers to mandatory access control, i.e. based on attributes that always exist on a mandatory basis, not provided at the user's discretion. Primary examples include the user's IP address and the time of access. The *authentication* stage implements the HTTP authentication framework, described in RFC 2617 [174]. The framework specifies a mechanism for challenge-response authentication to be performed between a client and a server. The *authorization* stage comes after the *authentication*. It is designed for making authorization decisions against the identity information obtained from the *authentication* stage.

After these security-related stages, the MIME type [175] for the requested resource is determined. The MIME type may need to be sent back in the response message to provide presentation hints for the client, e.g. an HTML page or an image. The *fixups* stage is reserved for future extensions that do not fit into the request handling cycle. After this stage, the actual response message is then generated and sent back to the client.

Apache then logs the processing of the request. This always happens after the response is sent, and may be optionally invoked at any other stages where logging is required. Finally, Apache cleans up all transient resources (e.g. allocated memory, open file handles, etc) and returns to the waiting state for another request.

For each stage, Apache goes through a chain of modules and sequentially invokes their handlers if they exist. If a module does not define a handler for a stage, it is simply ignored. Note that while the access control, authentication and authorization stages are intended for distinct purposes, there are few practical differences in the way they are treated by Apache.

6.2.3 Integrating Fidelis

An Apache module, `mod_fidelis`, has been implemented to provide Fidelis trust management for the Apache web server. At the heart of the module is a Fidelis policy engine which performs policy inference. The policy engine is invoked at various stages of Apache's request handling cycle to determine how a request should be processed. `mod_fidelis` also provides a parser for custom configuration directives, utilizing Apache's standard configuration parsing mechanism. The overall architecture is illustrated in Figure 6.5.

There are two types of configuration files in Apache: global and per-directory. The global configuration file is read and parsed when Apache starts up, while a per-directory one is parsed when a directory is accessed. A configuration file consists of a set of *directives*, which can be of per-server or per-directory scope. `mod_fidelis` defines custom directives for:

- the URL to a FPI (Fidelis Policy Interchange) document;

- the public key pair of the policy owner (i.e. the **self** principal); and
- default environment bindings.

`mod_fidelis` initializes the policy engine by loading the global policy document on a per-server basis. Subtrees of the document tree served by Apache may be made subject to different policies using per-directory configuration files. Policy files specified in a per-directory configuration file may replace or add onto the global, per-server ones, depending on need. The same applies to environment bindings.

`mod_fidelis` exposes the internals of Apache through Fidelis actions. It defines two types of actions: actions that represent hooks in a module and actions that correspond to HTTP methods. All these actions are defined to be simple actions, i.e. without parameters. The actions defined by `mod_fidelis` are summarized in the following table:

Hook actions	`translate`, `header-parsing`, `type-check`, `logging`
HTTP actions	`GET`, `POST`, `OPTIONS`, `HEAD`, `PUT`, `DELETE`, `TRACE`, `CONNECT`

When a request is being processed, `mod_fidelis` successively queries the policy engine. For example, at the URI translation stage, the module queries for the `translate` action; on the headers parsing stage, the `headers-parsing` action is queried; the MIME type checking stage follows a similar procedure. The exception is the security-related stages, in which case, the action representing the actual HTTP method will be queried for an authorization decision. For example, for a typical page request, the `GET` action will be queried, while for a form submission, the `POST` action will be queried instead.

Typical Fidelis applications would define parameters for actions and trust statements to carry more specific information. In `mod_fidelis`, a different approach is used. Environment variables are used to provide additional information regarding every request. When a request is received, these variables will be bound to values extracted from the request. The environment variables include the standard set of variables exposed to CGI programs by the server, e.g. `REMOTE_HOST` gives the host name of the web client, `PATH_INFO` gives the requested resource path, and `HTTP_USER_AGENT` identifies the web client. Figure 6.6 shows a set of commonly-used CGI variables as a reference. There are also some additional, non-CGI variables defined in the environment, such as `REQUEST_TIME`. Policies may modify the bindings to these variables, which would influence the way Apache handles a request. For example, a policy may redirect a request by modifying the value of `PATH_INFO` during policy evaluation. The reason for this unconventional approach to parameter handling is that the additional information is identical across all actions – derived directly from HTTP requests. Employing a uniform access mechanism is more convenient and less error-prone.

There is no default set of trust statements for `mod_fidelis`. It is up to the administrator to define trust statements that suit the application requirements using the standard mechanism provided by Fidelis. For the rest of this section, we shall examine the use of `mod_fidelis` in some particular application scenarios.

Example 6.4 Fidelis can replace the existing access control and authorization mechanism in Apache through a unified framework. Recall that the term "access control" in Apache refers to non-discretionary access control, and is often related to host name or IP-based authorization. An example of host-based authorization policy specified in Fidelis would be:

```
grants GET: p, POST: p
where PATH_INFO == "/" && REMOTE_USER = "elite.jesus.cam.ac.uk"
```

The above action policy gives access to the resource root ("/") using the HTTP GET or POST methods to clients on the host at `elite.jesus.cam.ac.uk`. Subnet addresses may be specified in a similar fashion as Apache. For example,

```
grants GET: p, POST: p
where PATH_INFO == "/internal" && REMOTE_USER = "128.232."
```

Environment variable	Description
`CONTENT_TYPE`	The MIME type of the query data (e.g. text/html)
`HTTP_USER_AGENT`	The web client the user is using (e.g. Mozilla/5.0 Galeon/1.2.1)
`PATH_INFO`	Extra virtual path information given by the client. From a URL, this is the path after the domain name.
`PATH_TRANSLATED`	The translated version of `PATH_INFO`, mapped into a physical pathname.
`REQUEST_METHOD`	The HTTP method used to make the request (e.g. GET, POST, etc.).
`REMOTE_HOST`	The domain name of the computer running the web client.
`REMOTE_ADDR`	As above, but in IP address.
`SERVER_PROTOCOL`	The protocol in use (e.g. HTTP/1.1)
`SERVER_NAME`	The host name of the computer on which the server is running.

Figure 6.6: Commonly-used CGI variables.

allows GET and POST requests on `/internal`, issued by any host within the subnet of 128.232. When a request is being processed, at the authorization stage of the processing cycle, `mod_fidelis` queries its policy engine for a decision. It first constructs an environment, binding environmental variables to their initial values. It then issues a query of the requested HTTP method. For example, if a web client requests a dynamic page at `/internal/member_data.php` using the POST method, a query for the POST action will be issued. Therefore, according to the action policy, if the requester resides within the 128.232. subset, the action will be authorized.

Suppose within an organizational intranet, the details of the company's account are published at `/internal/accounts`, and are strictly available only to accounts staff. Suppose every employee at the accounts department will be issued with a trust instance `as_accounts` asserting his or her role in the company. An action policy may therefore be written as follows:

```
as_accounts: self -> p
grants GET: p, POST: p
where PATH_INFO == "/internal/accounts"
```

This states that for a GET or POST method on `/internal/accounts`, its requester must be the subject of a valid `as_accounts` instance.

Example 6.5 One of the much-needed features for the WWW is the ability to filter content according to certain criteria, e.g. age, premium level, etc. This can be observed in the proliferation of parental control systems, such as CyberSitter™, SafeSurf™, and KidShield™. Most of such systems behave as a personal firewall, filtering the contents of web pages according to some criteria and heuristics as they are being received, e.g. scanning for certain keywords, comparing the address against a 'blacklist', or applying image recognition heuristics.

A more comprehensive filtering framework includes a *rating service*, which issues *rating labels* for sites (or pages). When a page is being requested, the filtering software first retrieves its label and makes a decision based on the information on the label. A label typically contains descriptive keywords indicating the nature of the content. Fidelis may be used to implement such a framework.

As an experiment, a `mod_fidelis`-enabled Apache server is configured as a HTTP proxy, acting as a filter for its clients. In this configuration, all client requests are forwarded through the proxy, which filters the response from the web server according to the policy specification. Suppose an imaginary

company, *CyberRating Inc.*, provides rating services for web sites and issues rating labels in the form of trust instances. It defines a trust statement, `rating`. An example `rating` instance is shown here:

```
rating("http://some-site.com/news.html", 0x3a81ba8, 3, 0, 0):
CRI -> some-site.com
```

where `http://some-site.com/news.html` gives the URL to the page, `0x3a81ba8` gives the cryptographic digest of the content, and the following three numbers give the level of violence, nudity and strong language on the scale of 0 to 5 (strongest). As a shorthand, the symbol '`CRI`' is used to represent the public key identifier for CyberRating Inc.

Action policies can then be set up on the proxy server to filter web pages. By default, GET or POST requests on a resource *without* an accompanying `rating` instance will be blocked. Moreover, conditions on parameters in `rating` instances may be specified to suit the desired level. For example, an action policy on the proxy might be:

```
CRI.rating (path, hash, violence, nudity, language): CRI -> q
grants GET: p, POST: p
where PATH_INFO == path && violence <= 2 && nudity <= 1
```

which states that in order for a page to be retrieved, it must be rated by CRI, with the level of violence and nudity less than 2 and 1 respectively. Note that we use a dot notation to indicate that `rating` is defined by CyberRating, rather than locally. The subject of `rating` is the site that hosts the page, and is normally different from the client principal. Two different variable placeholders are therefore used for the subject and the requester.

When the client makes a GET request, the request is sent to the proxy, which forwards the request to the ultimate web server. The web server then responds with the requested page. At this point, the proxy performs two operations. First, it attempts to retrieve the corresponding `rating` instance from CyberRating. If a valid `rating` instance is not available, it aborts the process. Otherwise, it queries the policy engine for an access decision with the collected `rating` instance.

Example 6.6 It is rapidly becoming a norm that a web user often needs to maintain multiple username/password pairs. While decentralization, autonomy and independent management are the key concept for the WWW, under this circumstance, it becomes a liability for users because of the inconvenience and operational overhead. *Single sign-on (SSO)* systems are introduced to address this problem. The idea is that a user only needs to authenticate once, and will then be able to access many sites without needing to re-authenticate at each site. Many commercial solutions exist, with the leading ones including Microsoft Passport™[176], Entrust GetAccess™[177], and RSA ClearTrust™[178].

Fidelis is naturally suited to implementing single sign-on because of its inherently decentralized nature. The mechanism centres around a time-bound trust instance that is issued to a user once he/she is authenticated at a site. The trust instance proves the holder as a valid user. The user may then present the trust instance to participating sites for access. As a demonstration suppose an imaginary company, SSO Technology Inc. (hereafter referred to as "SSOTech") offers a single sign-on authentication service. We set up a web server for SSOTech with `mod_fidelis` support, where a login page for user authentication is served. Suppose SSOTech defines a trust statement `authenticated` that carries two parameters, a unique identifier and a premium level (assuming SSOTech offers three levels of premium access, 0 to 2). An example trust instance looks like:

```
authenticated("DX4019169", 2): SSO -> Alice
```

Sites using SSOTech's service will simply need to identify instances of `authenticated` trust statements. Suppose a fake online entertainment site, *entertainmentoday.com*, uses the service. It allows all authenticated users to access the member area and all privileged users (with the premium level of 1 or above) to access the privileged area. The action policies could be written as:

```
SSO.authenticated(id, level): SSO -> p
grants GET: p, POST: p
where PATH_INFO == "/member"

SSO.authenticated(id, level): SSO -> p
grants GET: p, POST: p
where PATH_INFO == "/member/privileged" && level >= 1
```

A user visiting the site would first need to obtain an `authenticated` trust instance from SSOTech, either directly by authenticating at SSOTech's login page, or through entertainmentoday.com's web gateway to SSOTech's server. The trust instance may then be presented to gain access to entertainmentoday.com and other participating sites.

The implication of single sign-on is that the authentication process is delegated to a third party. In this example, SSOTech has the authority to decide its means of authentication, e.g. passwords, or digital certificates, and participating sites are expected to trust the strength and security of its authentication scheme. If a higher level of assurance is desired, a participating site may request additional trust instances to gain access using standard Fidelis mechanisms.

6.2.4 Discussion

The two major advantages of integrating Fidelis with the WWW are increased flexibility and enabling decentralized management for web-based applications. With Fidelis, complex access policies may be specified. In previous examples, mandatory, role-based and trust-based access control policies have been implemented. The flexibility is mostly due to the strong policy support of Fidelis, combined with appropriate interfaces to the Apache server. Decentralized management is a direct result of applying Fidelis, where the support for autonomous and interworking services is the fundamental notion. This is seen in the single sign-on example, where the management of authentication and authorization is clearly and securely separated.

On the other hand, while the architectural design satisfies the transparency requirement, the smart proxy introduces an additional layer. Ideally, Fidelis should be integrated with web browsers, with the advantages of increased performance, better security, and avoidance of an additional architectural component in the request/reply chain.

6.3 Case study: an electronic marketplace

A phenomenon facilitated by the World Wide Web is an ability for strangers to conduct business transactions online, resulting in the rapid boom of *electronic commerce* over the past few years. The aim of this case study is to provide a simulated study for the use of trust management, and specifically Fidelis, in an electronic commerce setting. While this case study is not based on a real online business, attempts have been made to closely model the actual operations and interactions between businesses and consumers to provide a realistic study.

6.3.1 Background

We focus our study on *electronic marketplaces*. The concept of an electronic marketplace is similar to traditional, physical markets where sellers and buyers aggregate, meet and carry out business. The basic idea is simple: with the large and fast growing consumer and supplier base, it is increasingly difficult for consumers to search and match their needs and for suppliers to be known and found by their potential customers. An electronic marketplace is intended to provide a central venue where suppliers gather to project a single, virtual shop offering combined ranges of products. Electronic marketplaces are rapidly gaining popularity, with prime leaders such as Yahoo! and Amazon. For example, Amazon started out as an online bookseller. However, over the years, it has gradually evolved into an electronic marketplace through partnership, offering items ranging from books to CDs/DVDs, consumer electronics, and houseware.

6.3.2 Environment

In this case study, we consider an imaginary electronic marketplace company called *virtua-marketplace.com*[2], a number of participating stores, consumers and an independent third-party, Better Business Bureau. The primary functions of these entities are summarized below:

virtua-marketplace.com

- *Portal contents.* Product advertisements from member stores are regularly collected, centrally stored and processed. Portal pages are generated to show a catalogue of available products.
- *Browsing/searching facility.* Buyers may browse or use the searching facility to locate products. For each product, a complete description is provided, together with some brief information about the sellers (stores).
- *Smart shopping.* A buyer may express his/her interest, preferences, or needs and let virtua-marketplace.com shop for the appropriate products/suppliers. For example, a consumer may express the willingness to spend at most £150 for a DVD player made by either SONY, Pioneer, or Phillips, with at least 2 years of warranty. This facility is available for premium members or for a service fee on a per-use basis.
- *Transaction agent.* For a commission, virtua-marketplace.com can deal with transactions on behalf of member stores or consumers. This is useful where either party wishes to engage in a pseudonymous transaction where real identity cannot be traced under normal circumstance. Note that virtua-marketplace.com is assumed to be legally obliged to protect this identity information.

Stores

- *Managing the collection of goods/products.* A store may maintain a database of products, where each product has an entry consisting of a description, a specification, a stock count and some additional notes (e.g. on promotion or sale).
- *Advertising.* When new products are introduced, a store may choose to notify virtua-marketplace.com for the new arrival. This mechanism works in complement with virtua-marketplace.com's periodic polling method.
- *Management of product information.* A store may send active notification to virtua-marketplace.com when product information changes, e.g. changes of price, stock level, or additional notes.
- *Processing purchases.* A store may have the facility to process purchases directly with customers. This typically involves the payment process, stock check, scheduling for delivery, and establishing after-sale policies.

Better Business Bureau

- *Rating service.* It provides a credit rating for online stores. The rating of a store may be affected by the monitored performance of the store, by transaction history, or by comments from past customers.

Consumers

- *Browsing/searching the product catalogue.* The browsing and searching facility on virtua-marketplace.com is open to any web user, not just registered members.
- *Purchasing goods.* From a consumer's point of view, purchasing mainly involves filling in an online order form which includes payment details.
- *Reporting.* A consumer may comment on his/her experiences with an online store and submit this information to virtua-marketplace.com. The opinion will then be reviewed and verified, and may be used internally or forwarded to the Better Business Bureau.

[2]The domain names used in this case study are non-existent at the time of writing.

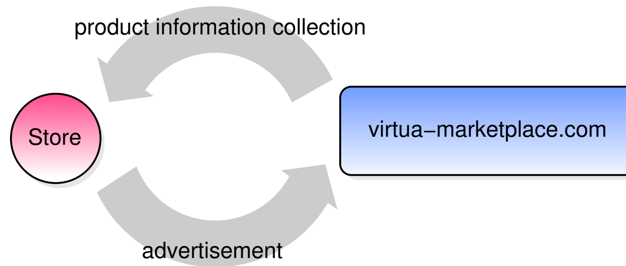


Figure 6.7: Incorporating product information

- *Recommendation.* Consumers may recommend online stores or goods to one another.

The case study builds on top of the web service architecture described in Chapter 4 and the WWW integration of Fidelis described in the previous section. The main interface of virtua-marketplace.com is a portal built in standard HTML and PHP4 [179]. PHP4 is an open-source, server-side scripting language, allowing, for example, contents to be dynamically generated from databases. Interactions between the entities described above are implemented mainly using web service interfaces.

6.3.3 Membership management

virtua-marketplace.com employs a subscription scheme for both stores and consumers. A store must subscribe to be able to advertise and submit new products to virtua-marketplace.com's product database. Subscribed stores also have the benefits of directed marketing, where virtua-marketplace.com analyzes its consumer base and periodically recommends stores of interest to customers. Subscription for consumers is intended to maximize their ease of use. A subscribed consumer may be given a special offer from time to time, and use advanced services including smart shopping.

virtua-marketplace.com applies role-based modelling for these two kinds of subscribers. Two trust statements are designed to represent the membership of a business user and a consumer, `as_business` and `as_consumer` respectively. `as_business` has one parameter, the URL of the online store. An example instance is:

```
VM.as_business("www.buysportstuff.com"): vm.com -> 0xba2d54f...
```

Such trust instance expresses two meanings. First, the subject (in this case, `0xba2d54f...`) is a business subscriber of virtua-marketplace.com. Second, the subject is the online store at the named URL, (`www.buysportstuff.com`). Note that the dot notation is used to indicate that `as_business` is defined within the scope of virtua-marketplace.com (`VM`), and the symbol `vm.com` represents the public key identifier for virtua-marketplace.com, and will be used consistently throughout this case study.

`as_consumer` is similar to `as_business` but slightly simpler as it contains no parameter. An instance of `as_consumer` conveys the simple message that the subject is a subscribed consumer of virtua-marketplace.com.

6.3.4 Product catalogue management

virtua-marketplace.com maintains the product catalogue in a database. The database contains information about products available at its member stores. For each product, there exists in the catalogue an entry consisting of the description, the price, the stock level, and a textual field for additional notes. Each store may optionally maintain a local database for a similar purpose. As shown in Figure 6.7, the database at virtua-marketplace.com is populated by two mechanisms : *catalogue collection* and *advertisement*.

Catalogue collection is a pull mechanism. When a store first registers with virtua-marketplace.com, its full catalogue is retrieved and incorporated. Subsequently, virtua-marketplace.com periodically collects product information at the frequency specified by each member store.

Advertisement is an active push mechanism. A store may send product range updates to virtua-marketplace.com when new products arrive. virtua-marketplace.com employs Fidelis action policies to protect the active interface. The advertisement interface at virtua-marketplace.com is abstracted as an action, `advertise`, and the advertisement policy states that advertisement is accepted if and only if it is originated from a business user (i.e. an online store). Expressed in Fidelis,

```
as_business(url): self -> p grants advertise(product_details): p
```

When new product information is incorporated, either from the catalogue collection or through advertising, virtua-marketplace.com allocates a product identifier and issues a `product_store` instance to the store that offers the product. The product identifier is used internally to help produce the catalogue portal at virtua-marketplace.com. The `product_store` instance is for identifying the *owner* of the product entry in the catalogue database. One use of this trust statement is for product information update. Product information on the catalogue at virtua-marketplace.com is only allowed to be modified by the owner of the product entry. This is requested when, e.g. there is a price change or stock level change. The action policy is:

```
as_business(url): self -> p, product_store(product_id): self -> p
grants update_product(new_details): p
```

This policy not only requires a `product_store` instance, but also a matching `as_business` instance to authorize update on a product entry. This may seem redundant but gives tighter security as it places an explicit requirement that the subject of the presented `product_store` instance must be a subscribed business user.

6.3.5 Reputation management

An important element ensuring the functioning of the electronic marketplace is a mechanism that enables unfamiliar parties to build trust and interact. While big players will benefit from brand recognition, small sellers must rely on other means to gain trust from potential customers. Reciprocally, while risking violating privacy, under some special circumstances, a seller may also wish to find out the credibility of a potential customer, e.g. to prevent fraud by repeating cheaters.

virtua-marketplace.com employs a simple reputation system where a rating for subscribers may be queried. The reputation system consists of two main sub-systems: an *opinion collector* and a *rating aggregator*. The purpose of the opinion collector is to gather feedback about interactions between subscribers (either store-consumer, consumer-consumer or store-store). It offers both passive and active mechanisms. The passive approach is a reporting mechanism, where transaction experiences may be given in free-form text, and some evidence (such as transaction record, payment evidence, etc) may be attached to support the case.

The active approach is based on a monitoring mechanism, and is designed for situations where virtua-marketplace.com is acting as an transaction agent on behalf of a store or a consumer. When virtua-marketplace.com is empowered with the task of executing a transaction, it is able to monitor whether the other party duly fulfills its duties. More details are described in the next section.

The rating aggregator is in charge of computing a rating value from collected opinions. For simplicity, the computation uses a simple average function, resulting a discrete value on the scale of 1 to 5. In this case study, submitted opinions are not verified, but in practice, it is important to guard against false reports.

The opinion collector may also take the rating from the Better Business Bureau as an input. The Better Business Bureau defines a trust statement, `rating`, whose instance gives the rating information of the subject. The `rating` trust statement has a single parameter, the rating value. The rating value is on the same scale of 1 to 5 as used in virtua-marketplace.com. This is intentional, as it simplifies the

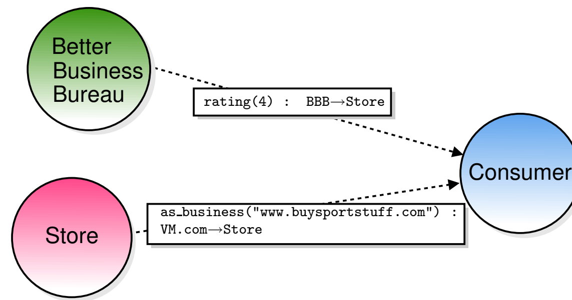


Figure 6.8: Supporting purchase decision.

design of the case study. The Better Business Bureau also defines another trust statement, `business`, which is intended to identify a public key as a business. The `business` trust statement includes a single parameter, the URL of the online business.

In the next section, examples illustrating the use of these trust statements will be provided.

6.3.6 Transaction processing: purchases

We shall consider the most common type of transactions in our setting – the purchase of goods. In many situations, a purchase is a direct transaction between the store and the consumer. Once a consumer finds the appropriate product and decides to proceed with a purchase, he/she places an order by filling in a form at the store's site.

Suppose the store is new to the consumer and, as a result, the consumer wishes to gain access to the credibility of the store before proceeding with the purchase. One approach is to make use of the rating information provided by the Better Business Bureau. The consumer may express a policy whereby a purchase may be initiated only if it is with a store which is a member store of virtua-marketplace.com and is accredited by the Better Business Bureau with a rating of 4 or above. The policy can be expressed as follows:

```
VM.as_business(url) : self -> p,
BBB.business(url) : BBB -> q, BBB.rating(r) : BBB -> q
grants purchase(product_id) : self
where r >= 4
```

Under this policy, for a purchase decision to be made, the store must provide the consumer with a valid instance of `VM.as_business`, which asserts that it is a member store of virtua-marketplace.com. Additionally, the consumer obtains the `BBB.rating` instance regarding the store from the Better Business Bureau and the rating value must be 4 or greater. Note that since it is possible that the same online store is known differently (i.e. different public keys) at virtua-marketplace.com and Better Business Bureau, matching is based on the URL of the store, instead of the subject public keys. Figure 6.8 illustrates the purchase scenario.

virtua-marketplace.com provides a service whereby it carries out purchase transactions on behalf of consumers. The design of the service has two advantages. First, it brings increased convenience to subscribed consumers, as the payment option (e.g. the credit card information) needs only supplied once. Second, it provides an opportunity to monitor the progress of the transaction, so that the credibility of both the buyer and the seller can be assessed. In addition to the two reasons, it also offers a possibility to implement *pseudonymous transactions*, which break the linkage from a transaction to other information about the consumer (e.g. buying patterns, personal interest, or credit information). Note that facilitating pseudonymous transactions is not the aim of this example but is mentioned here to highlight the potential of intermediaries.

Figure 6.9: Delegated purchase

For this service, virtua-marketplace.com specifies a trust statement, `delegated_purchase`. A consumer who wishes to make a purchase through virtua-marketplace.com would need to create an instance of `delegated_purchase` and pass it to virtua-marketplace.com. The trust instance contains details about the purchase, and for simplicity, it is implemented in this case study to carry a single parameter, the product identifier. Once virtua-marketplace.com receives the request, it begins the purchase procedure with the store. As part of the procedure, the store requires the `delegated_purchase` instance to be forwarded as a proof of authority. This is enforced by specifying the following action policy:

```
VM.delegated_purchase(product_id) : p -> vm.com,
grants purchase(product_id) : vm.com
```

The transaction continues if and only if the `purchase` action (locally defined within the scope of the store) is granted. This action policy only deals with delegation transactions with virtua-marketplace.com as the intermediary. There typically exists action policies handling other types of transactions, e.g. direct transaction with consumers. Note that in this policy, the variable placeholder `p` gives the public key identifier for the consumer who initiated the purchase. This approach therefore is not sufficient for hiding identities. Delegated purchase is illustrated in Figure 6.9.

6.3.7 Discussion

Unlike applications in previous sections, this case study demonstrated decentralized management, which is crucial for the highly distributed nature of the web. Each entity in the environment, virtua-marketplace.com, member stores, Better Business Bureau and consumers, may define their own trust statements and policies using them. Furthermore, policies defined by a party often depend on trust instances issued by other parties. For example, a consumer may rely on the information given by the Better Business Bureau to make purchase decisions. This mechanism facilitates the linkage between independently administered sites. Fidelis thus shows its potential in supporting large, decentralized applications.

Several aspects in the case study are intentionally simplified to allow us to focus on trust management problems. These include: catalogue database design, reputation computation, and transaction procedure. In real life, these issues must be given more comprehensive treatment. For example, reputation computation as implemented is based on an average function, whereas in practice, the function may need to take into account the transaction value, quantified risks and legal obligation, etc. Moreover, the transaction procedure needs to include some online payment scheme such as PayPal™[180].

6.4 Summary

In this chapter, we have examined three applications of Fidelis in detail so as to evaluate its effectiveness and practicality.

Role-based access control has been highlighted as a promising mechanism for new applications, addressing many inherent limitations of traditional access control schemes. Fidelis has been shown to provide RBAC functionality through the use of action policies and trust statements. This indicates that Fidelis may be employed as a general mechanism for access control.

We have also examined trust management in the context of the WWW. As an enabling technology, an Apache module with an integrated Fidelis engine has been implemented: `mod_fidelis`. This brings native trust management support to the Apache web server and allows us to carry out experiments with Fidelis. Several small experiments have consequently been constructed to implement different styles of authorization schemes.

This chapter closes with a case study of an electronic marketplace, whereby multiple parties participate and interact. This case study combines the use of Fidelis in both the web services and the WWW context. It has shown as a proof-of-concept that Fidelis supports decentralized management and therefore has the potential to be deployed for large-scale, distributed Internet applications.

In the next chapter, we provide a detailed analysis of Fidelis, evaluated against the research goals set out in Chapter 1 of the thesis.

7 Discussion

Discussion on the approach presented in this thesis has been given previously where appropriate. It is nevertheless important to evaluate Fidelis as a whole, analyzing it against the research issues described in Section 1.3, which are summarized below:

- Policy framework
- Managing scalability
- Decentralized collaboration among unfamiliar parties
- Privacy
- New approaches to decentralization

In this chapter, the aspects of Fidelis addressing each of the above research issues will be discussed in depth. Issues that are not addressed fully in this thesis will be highlighted, along with some discussion on possible future research directions.

7.1 Policy framework

A major design difference of Fidelis from other existing trust management systems is its strong emphasis on policy support. It features a comprehensive policy framework, backed by a clearly defined policy language – the *Fidelis Policy Language*. The policy language is abstractly specified, with the intention to allow various instantiations of the language to suit different application needs. More precisely, it intentionally does not include a type system, nor special sub-languages for assignment and conditional expressions.

One such instantiation is demonstrated in Section 4.3, in the form of the *Fidelis Policy Interchange*. Fidelis Policy Interchange is an instantiation specifically designed to facilitate policy exchange between web services. It is built upon XML [149] technologies, and adds the type system in the standard XML/Schema [159, 160] to the policy framework. It also introduces an extensible framework that allows assignment and/or conditional expressions to be specified in any agreed language. By default, it supports the XPath 2 [167] expression language.

A full analysis of the policy framework was given in Section 3.5.9, Chapter 3. Based on the analysis given there, and additional observations gained from constructing applications, the policy framework will be discussed in the following areas: expressive power, ease of use, ease of implementation, and runtime efficiency.

The Fidelis Policy Language (FPL) is designed to express two kinds of policies: trust policies and action policies. Aside from the syntactic sugar, in essence the language is based on first-order logic. There are two other features in FPL that increase its expressive power, namely the constructs of group and threshold principal, and the addition of negative statements (**without** clauses). These features are incorporated with the specific intention of capturing commonly-found real-world policies. Also important in our design is the flexibility of choice for sub-languages used in assignment/conditional

expressions. The choice of expression language inevitably affects the expressive power of the instantiated language. It was therefore decided to leave the decision to the individual applications. In summary, while a more rigorous analysis would be desirable, based on the experiences learned through the examples provided throughout the thesis, FPL may be considered to be sufficiently expressive for many uses.

On the ease of use of the Fidelis Policy Language, the language is designed to feature a clear syntax, with minimal use of symbolic operators to increase readability. In theory, it should be relatively easy for people with some computing background to understand and write policies. However, the language is not intended for non-expert use. As previously mentioned, the language needs to be instantiated when used in practice. One particular "instantiation" may be in the form of a GUI (Graphical User Interface) tool that employs the model that underlies the language as a foundation and presents user-friendly interfaces for the specification of policies. While the policy language is not ideal for non-experts, with its clear syntax and well-defined constructs, it is still arguably easier than writing "policy programs" in real programming languages.

On the issue of ease of implementation, the most crucial part of the policy framework is the trust management engine, where trust computation is performed. It is therefore sensible to restrict ourselves to considering the degree of difficulty in implementing the inference algorithm, without considering "boilerplate" code for parsing, decoding credentials, performing cryptographic operations, etc. In the demonstration implementation, the inference algorithm described in Section 5.1 is implemented in under 800 lines of C code. The C language is chosen mainly because of the convenience of integrating with existing software such as the Apache web server. The algorithm is implemented as a state machine with stacks. The core stack machine is implemented in slightly over 500 lines of code. The implementation took under two man-days to complete and test, with additional minor bug fixes. Based on this implementation experience, it may be safely stated that the policy framework is straight forward to implement. It should be noted however, that the algorithm in Section 5.1 is not the only algorithm that can realize the policy semantics. More efficient or optimized algorithms may require greater implementation efforts.

As previously discussed in Section 5.1.4, the algorithm has a worst-case runtime of $O(MN)$. where M is the maximum number of parameters for any trust instance, and N is the number of terms in all policies. While this is polynomially efficient, the value N would typically be large. It is envisaged that with appropriate scheduling and optimization, the worst-case runtime may be lowered. However note that the non-monotonic nature of the language has an impact on the runtime efficiency. It remains a research issue to study trade-offs between the runtime efficiency and expressive power, given the inclusion of negative trust statements.

7.2 Managing scalability

As discussed in Chapter 1, today's distributed applications are more demanding in terms of their scalability requirements as Internet-scale connectivity is now standard. The design of Fidelis is intended to meet these stringent scalability requirements. Well-respected principles in distributed systems are carefully examined and incorporated in its design, resulting in a fully decentralized architecture. Important features and design principles of Fidelis that increase scalability are to be examined in this section. While decentralization is a key to infrastructure scalability, management scalability still needs to be addressed, especially when the user/resource set is becoming large – likely for many new distributed applications. We shall therefore also discuss provisions and potential techniques in Fidelis addressing management scalability.

Similar to other capability-style authorization management systems, the key notion of scalability is *decentralized enforcement* of policies. Authorization in typical capability systems depends primarily on capabilities presented by requesters. The key advantage is that the point of enforcement need not have any knowledge of the requesters, thus allowing its administration to be separated from the policy authority.

Fidelis is designed to specifically allow decentralized enforcement. There are two aspects that

contribute towards this aim. First, the authorization model respects full local autonomy. Every principal (including services, hosts, and sites) is fully autonomous, with the discretionary power to design its policies. A principal is expected to only consult its own policies in making authorization decisions. Second, the validity semantics attempts to break dependency between the issuer and the acceptor of a trust instance under most circumstances. As described in Section 3.3.2, the fundamental concept behind the validity semantics is the determinism principle, whereby once the validity of a trust instance is *guaranteed*, it cannot be reverted. A guarantee usually is given as absolute time-bounds, with the only exception being the online status check, in which case, dependency between the issuer and the acceptor does exist. This is a trade-off between the degree of decentralization and timeliness requirement, and can only be judged at the application level.

Local autonomy has a greater implication than decentralized enforcement. As a general principle in distributed systems, localization is often regarded as an approach to increase manageability. This is particularly the case in Fidelis, where each principal has the freedom to design its own policies, define and specify its local trust statements and, furthermore, implement and enforce its policies. Every principal is conceptually responsible for issuing trust instances of its local trust statements. This level of autonomy is especially important in today's widely distributed systems because of the difficulties of having global authorities.

In most current trust management systems, creating and issuing credentials is often manual processes, usually requiring human intervention. As the user base grows, these manual tasks become a limiting factor. A solution is to integrate role-based access control into the trust management framework, thus users are treated as roles, and credentials are issued to users according to their roles. However, through the development of Fidelis, it became clear that the support for roles does not need to be an integral part of the framework for both simplicity and flexibility reasons, since the functionality may be supported through specialized policies, as demonstrated in Section 6.1. If required by applications, meta-policies could easily be written to control the behaviours of the issuance of trust instances, thus addressing management scalability problems.

7.3 Decentralized collaboration

In Section 1.3, the focus of the description on decentralized collaboration is on collaboration among strangers. In this section, however, we shall examine support in Fidelis for collaboration both among mutually known parties and among strangers.

In a collaborative environment, complex authorization problems arise. In traditional approaches, participants of a collaboration often need to know in advance about each other in order to attach appropriate authorization policies. Such approaches fall short of ideal in a decentralized environment, where strangers may participate in collaboration and/or the number of participants may be too large for the individual specification of authorization policies to be practical. One approach that simplifies decentralized authorization in collaborative environments is *attribute-based authorization* [103].

The basic concept behind attribute-based authorization is that certified attributes are trusted as the basis for making authorization decisions. In its basic form, a principal only needs to recognize attributes that it certifies. In Fidelis, this corresponds to the concept of recognizing locally-defined trust statements in trust/action policies. The use of attributes by itself does not solve authorization problems in collaborative environments. However, it provides a foundation for decentralization.

Attribute-based authorization may be extended to allow a principal to make use of attributes certified by others. In Fidelis, this equates to the notion of recognizing trust instances issued by third parties in policies, effectively establishing an explicit trust relationship between the local principal and the trust instance issuers. The key advantage is that it allows *chaining* of principals: principal A may recognize a trust instance issued by principal B. B issues the trust instance because it recognizes a trust instance issued by C, and so on. With principal chaining, collaborative authorization is significantly simplified, provided appropriate "third-party" principals are introduced. Some further discussion will be given later in Section 7.5.

With this extension, third-party principals were identified by their identities. It is possible to

allow further decentralization by recognizing third-party principals by their attributes. For example, a cinema may sell discount tickets to people who possess student cards, which are issued by some educational entity, and the cinema recognizes education entities if they are certified by the Education Authority. Fidelis also supports this type of policy through its policy framework. For example, the cinema may express their policy as follows:

```
student_card(): x -> y, university(): EA -> x grants ...
```

in which case, the placeholder x is an unknown principal, but it is required to be certified as a university by the Education Authority (EA). This allows strangers to be identified and trust to be built on certificated attributes. With the powerful policy framework, Fidelis is capable of providing a comprehensive attribute-based scheme, enabling collaborative authorization.

A special case in collaborative environments is when two untrusted strangers attempt to form a collaboration. Due to mutual mistrust, it is often undesirable for either side to disclose sensitive knowledge in order to gain the trust of the other. However, in contradiction, policies may sometimes require a requester to disclose trust instances that contain sensitive information. Addressing this type of collaboration, a trust negotiation framework is designed for Fidelis. The framework includes a protocol that allows a pair of strangers to incrementally disclose trust instances based on the knowledge presented by the other party.

While the negotiation framework mainly presents a proof-of-concept design and much work remains to be done, it has demonstrated a crucial novelty in its approach to trust negotiation: the negotiation protocol is fully policy-driven, with the use of Fidelis to control the protocol behaviours. Comparing with other notable work in this area [168, 170, 181, 169], the policy-driven protocol is significantly more flexible: an application may have its own tailor-made negotiation protocols by simply standardizing on the set of meta-policies that control the protocol behaviour; protocol behaviours may be fine-tuned by encoding personal preferences in meta-policies; and new negotiation protocols may easily be tried in this framework. The downside is the lack of formal rigour, as the protocol tends to be over-flexible.

7.4 Privacy

As identified in Section 1.3, in practice privacy issues will play an important role in the public adoption of a trust management system. Generally speaking, there are two areas of concern that a trust management system should tackle. Firstly, credentials (in the case of Fidelis, trust instances) may contain sensitive information which should not be made publicly available. A principal using trust instances should only learn "just enough" information from them for its tasks. Secondly, the system should prevent linking of credential usage. Such linkage often reveals vital information regarding one's behaviour, living pattern, purchase preferences, etc. The system should enforce *unlinkability* where possible.

While the design of Fidelis does not directly address privacy issues per se, it nevertheless has an important provision for possible future work. Underlying the framework is its key-oriented nature, whereby every principal may generate its public key pair at any time. Public keys are used as principal identifiers, without linking to any property of the principal. A possible approach to provide unlinkability is therefore to require a principal to generate a fresh key pair on every use. Although such a scheme is effective, it unfortunately may not be practical in real applications, since a fresh public key provides no value for gaining trust.

Fidelis does not provide a solution for selective disclosure of parameters in trust instances. At present, data encoded in the Fidelis Interoperable Credential (FIC) format are in cleartext. However, recall that the conceptual framework of Fidelis does not impose restriction on how trust instances should be encoded. Other formats which support selective encryption may be used instead of FIC. Alternatively, FIC may be extended to integrate XML Encryption [182], which allows sections of any XML document to be encrypted under different keys. Furthermore, protocols may need to be developed to support decryption of parameters of trust instances.

7.5 Decentralization approaches

Similar to most other decentralized authorization systems, Fidelis supports the concept of *delegation of authority*. However, in most other systems, including PolicyMaker/KeyNote [9, 20], SDSI/SPKI [17, 18], and TrustEstablishment [101], the authority in their context refers to the authority to access. Fidelis supports the authority to grant access in the form of *action policies*. Additionally, Fidelis supports a different type of authority – the authority to assert attributes, and this is provided through *trust policies*. In this section, we will discuss some approaches to the structuring of authority based on the facilities in Fidelis and their relative pros and cons.

Hierarchy. Hierarchical structures are common in human societies, for example company structure, government structure, etc. In a hierarchy, there is usually clear separation (of responsibilities and authority) between levels, and typically, an entity mainly manages its direct subsidiaries but not further descendants. It is fairly straightforward to implement hierarchies in Fidelis, by restricting policies to recognize only trust statements defined by superior principals in a hierarchy.

With proper design and strict implementation, hierarchies can be an effective means of organizing a large number of principals. They simplify the enforcement of standards. For example, educational institutes under the Educational Authority may be directed to follow some standard for issuing student identity cards. Hierarchies are often relatively manageable, due to their centralized nature. However, due to their rigid structuring, hierarchies tend to be inflexible, especially when changes to the structure are to be made. Furthermore, the hierarchical approach is generally not feasible for large-scale systems, where global standards on hierarchy structures are difficult to agree.

Peer-to-peer (P2P). On the other extreme to hierarchies, another style is to impose no constraints on the structure at all. As principals in Fidelis are all treated equally, and may freely interconnect, any pair of principals may establish a local service-level agreement (SLA) that details the trust statements and actions agreed by both parties, and may additionally include associated policies. A principal should then follow the directives set out in the SLA and implement the semantics of the trust statements, actions and policies.

Peer-to-peer structuring is suitable for applications where loose relationships exist among principals, and is ideal when trust relationships are dynamic and constantly changing. It is also useful for applications where local agreement is sufficient for authorization needs, without requiring complex hierarchies. Notable applications include file sharing programs, messaging applications, and trading platforms.

There are two major advantages of this approach: flexibility and scalability. The notion of SLAs is naturally pairwise[1]. Changes to a SLA therefore only involve the two parties that agreed on that SLA. For this reason, P2P structuring is more flexible, and easier to modify. P2P structuring is self-managed in the sense that every principal maintains its own SLAs. This fully decentralizes the management tasks, and hence achieves greater scalability. However, self-management may also be considered as a drawback, as it adds management burdens to every principal. Moreover, the lack of structure in the P2P approach may be undesirable for some applications as it gives little authority of control.

Hybrid. The two structuring approaches discussed so far both have their merits and weaknesses. Based on the examples and case studies in this thesis, it is observed that a combined approach often gives a satisfactory balance. For example, within an organization, in order to simplify management tasks, a local hierarchy may be imposed, e.g. headquarters, divisions, regional offices, departments, etc. However, the organization may enter SLAs when collaborating with other business entities, representing the organization as a whole.

This combined approach approximates the real-world more closely than the previous two approaches, and maintains a reasonable level of control, structuring flexibility and scalability.

[1] Although pairwise SLA is not strictly required, the set of participants agreeing on a SLA is usually small.

Nevertheless, these advantages greatly depend on the design of the structure. Improper design may lead to combined disadvantages of the two approaches, rather than advantages.

In Fidelis, the decentralization approaches discussed above may be explicitly expressed within its policy framework, whereas in other decentralized authorization systems, structuring is typically done in an ad-hoc manner, without support from formal policies. The Fidelis policy-driven approach simplifies the verification and implementation of the structuring design, as the structure is encoded in the form of Fidelis policies. From this perspective, Fidelis may be considered a more comprehensive platform for decentralized applications.

7.6 Summary

This chapter provides a critical discussion of Fidelis as a means of its evaluation. The discussion is organized to reflect the research issues described in Section 1.3, and examine each issue in depth. The next chapter concludes this thesis by providing a summary of the contributions and some directions for future research.

8 Conclusions and Future Work

Future distributed applications will be of vast scale, widely open, and will often need to deal with complex collaborative interactions. A key necessity for the development of these applications will be a powerful, scalable, flexible and extensible authorization management framework. This thesis reviewed the state-of-the-art in this area, examined and identified research issues that are yet to be addressed. It is the conclusion of this work that a suitable authorization management framework for the emerging distributed applications must possess as a minimum:

- A highly decentralized architecture
- A comprehensive policy framework

To satisfy the above criteria, a novel trust management framework, Fidelis, has been designed and implemented as part of the work presented in this thesis, addressing many of the identified research issues. Its evaluation has been provided through the implementation of several examples, applications and case studies, and has been shown to be a promising authorization framework for future applications.

This chapter concludes this thesis. Section 8.1 highlights the main contributions of this work. Section 8.2 suggests some future directions that may be undertaken to further enhance Fidelis. Section 8.3 provides a closing remark on this thesis.

8.1 Summary of contributions

The main contribution of this thesis is the proposal of a policy-driven, decentralized trust management framework – *Fidelis.* As a recapitulation of Section 1.4, through the conceptualization, design and implementation of Fidelis, the following contributions have been made:

- Proposing the *trust conveyance model* as a generic model, serving as a simple foundation for future trust management systems.
- Designing a powerful policy framework, realizing the trust conveyance model and allowing complex security policies to be expressed under a unified framework.
- Designing and implementing Fidelis, thus providing an infrastructure on which future web service applications can be built.
- Designing and implementing an algorithm for computing trust compliance. The algorithm is designed to demonstrate the feasibility of Fidelis and its policy framework.
- Proposing a policy-driven trust negotiation framework, which enables collaborative strangers to gradually disclose sensitive trust instances and learn about each other.
- Evaluating Fidelis in several application contexts. While the focus of these studies has been on Fidelis itself, the experiences learned may indeed be useful to other systems.

8.2 Future work

This section suggests some selected future work. We shall be discussing four main areas: trust metrics, privacy support, trust compliance algorithms, and trust negotiation frameworks.

Recently, in contrast with the trust management approach, a distinct but complementary approach based on the so-called *trust metrics* (or "trust models" in some terminology) to deal with uncertainty has gradually become an interest in the research community. As briefly discussed in Section 3.3.3, the basic idea is to derive a *trust value* for a principal, based on several factors, such as past record, quantified reputation, quantified risk, etc. The value can then be used as a basis to predict the future behaviour of the principal, within some acceptable error. An extension of Fidelis would be to integrate these trust metrics, providing solutions where a fuzzy notion of trust is preferred or required. A possible integration approach would be to specify standards for trust statements that carry parameters for holding trust values. Principals complying with the standard should then compute and interpret the trust value following the specified trust metrics. Policies may be written to selectively grant or deny access based on trust values in trust instances.

As already discussed in Section 7.4, privacy issues have not been directly addressed in Fidelis. There are two major areas wherein Fidelis should be extended to provide enhanced privacy support. Firstly, in order to protect sensitive parameters (i.e. attributes) in trust instances, trust instances should be encrypted and cryptographic protocols such as SSL/TLS [164] should be employed to ensure sensitive trust instances are only exposed to the intended parties. However, encrypting the entire trust instance is often overkill. Ideally, selective encryption on certain parameters is desirable. Secondly, usage of trust instances, when linked, may provide an insight about one's behaviour. De-identification is the typical approach to prevent unlawful linkage of data. While in Fidelis, principal identifiers are simply public keys, providing no identification in themselves, however, when used in trust instances, a linkage is formed and may therefore allow identification. There is some interesting research in this space, notably Brands' Digital Credentials [183].

The third area of future work is on the trust compliance algorithm of Fidelis. Chapter 5 described an algorithm that implements the policy evaluation semantics defined in Section 3.5.8. However, the algorithm is intended as a proof-of-concept, and as a result, correctness is of a higher priority than efficiency. Some possible work on this includes: designing an efficient algorithm that is formally proved to faithfully implement the defined semantics, techniques for compiling policies and applying optimizations on the compiled policies, possibly by precomputing possible evaluation paths, and designing distributed algorithms to perform the policy computation.

The last area that may be explored further is the trust negotiation framework. The negotiation framework described in Section 5.2 represents the first attempt at applying Fidelis to deal with situations where strangers are involved. While the correctness of the negotiation protocol has been demonstrated by implementation, formal proof is required to study properties of the protocol, including termination and the states of both principals after protocol runs. As it currently stands, it serves mainly as a research framework for designing and experimenting with new protocols.

8.3 Conclusion

This thesis has presented *Fidelis*, a fully policy-driven trust management framework, designed for widely-distributed Internet applications. The crucial novelty lies in its extensive policy support, which enables complex real-world trust-related policies to be expressed and enforced. Although much future research remains to be done, as studied and demonstrated in this thesis, we believe that the policy-driven approach adopted by Fidelis is the way forward in future research on trust management frameworks.

Action
An abstraction for a well-defined computation that may be subject to policy control. A common example of actions is a permission in an access control system, e.g. `read` access. An action may also encapsulate an access request, e.g. `query_balance (account)`.

Action policy
A rule specifying the conditions under which a requested action may be granted. In the Fidelis policy language, the conditions may include the presence and/or absence of certain trust instances and/or contextual constraints.

Assertion
(In the context of a trust statement) a belief, a claim, or a fact regarding a principal (i.e. the subject), stated or declared by another (i.e. the truster).

Context
(In the Fidelis policy framework) The situational conditions under which the interpretation of an assertion is consistent with its intended meaning given by its truster.

Conveyance source
In an instance of trust conveyance, the principal who provides the trust statement for transfer. Effectively, the conveyance source provides its knowledge for other principals (i.e. the conveyance targets).

Conveyance target
In an instance of trust conveyance, the principal who receives the trust statement from the conveyance source. The conveyance target collects new knowledge from the conveyance source.

Distrust
The opposite notion to *trust*. Distrust refers to a set of *negative* assertions that a principal holds with regard to another principal. Note that this definition is distinctively different from the absence of trust.

Principal
A principal in Fidelis is an entity which has control over a public key pair, i.e. the principal *speaks for* the key.

Target
The shorthand for *conveyance target*.

Trust
A set of *positive* assertions that a principal holds with regard to another principal. It typically represents one's knowledge, beliefs or claims about another principal in some context. Such information is abstract, and is expected to be embodied through *trust statements*.

Trust conveyance
The process of transferring a trust statement from one principal to another. This transfer models the mechanism of *knowledge-passing* in daily life, whereby a principal spreads its knowledge to others. This term is chosen to reflect the fact that a trust statement contains the trust information asserted by the truster, thus passing a trust statement effectively conveys trust information.

Trust instance
The short name for *trust statement instance.*

Trust policy
A rule specifying the conditions under which a new trust statement may be issued. A trust policy formalizes one's process of trust establishment with others. In the Fidelis policy language, the conditions may include the presence and/or absence of certain trust instances and/or contextual constraints.

Trust specification
The short name for *trust statement specification.*

Trust statement
A digitally signed credential that acts as the basic building block in the Fidelis policy framework. A trust statement includes the truster who issued the trust statement, the subject who the trust statement is in respect of, a set of assertions and a validity condition. The information contained in a trust statement represents the truster's *trust* (see the definition above) in the subject (under the interpretation of the intended context).

Trust statement instance
Equivalent to *trust statement.* This term is introduced for use in situations where the clear distinction between the *specification* and *instances* of trust statements is essential. It is used extensively in the description of the Fidelis policy language.

Trust statement specification
One component of the trust statement is a set of assertions. A trust statement specification defines the structures and meanings which the assertions follow. For example, in the Fidelis policy language (where an assertion is given as an attribute), a trust statement specification specifies the data type and provides an interpretation for the list of attributes.

Truster
Relative to a trust statement, the issuer of the trust statement. The term emphasises the fact that the trust statement contains assertions made by the issuer, and assertions in Fidelis are treated as trust information.

Source
The shorthand for *conveyance source.*

Subject
Relative to a trust statement, the principal to which the trust statement relates.

Bibliography

[1] J. Hine, W. Yao, J. Bacon, and K. Moody, "An architecture for distributed OASIS services," in *Middleware 2000 (Palisades, NY, April 4–8)*, no. 1795 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 104–120, Springer-Verlag, April 2000.

[2] J. Bacon, A. Hombrecher, C. Ma, K. Moody, and W. Yao, "Event storage and federation using ODMG," in *Proc. 9th International Workshop on Persistent Object Systems (POS9, Lillehammer, Norway Sept. 6–8)*, no. 2135 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 265–281, Sept. 2000.

[3] W. Yao, K. Moody, and J. Bacon, "A model of OASIS role-based access control and its support for active security," in *Sixth ACM Symposium on Access Control Models and Technologies (SACMAT 2001, Chantilly, VA, May 3–4)*, (New York, NY), pp. 171–181, ACM Press, May 2001.

[4] J. Bacon, K. Moody, and W. Yao, "Access control and trust in the use of widely distributed services," in *Middleware 2001*, no. 2218 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 300–315, Springer-Verlag, 2001.

[5] J. Bacon, K. Moody, and W. Yao, "A model of OASIS role-based access control and its support for active security," *ACM Transactions on Information and System Security*, vol. 5, Nov. 2002. To appear.

[6] ITU-T (Telecommunication Standardization Sector, International Telecommunication Union), Geneva, Switzerland, *ITU-T Recommendation X.509: The Directory – Public-Key and Attribute Certificate Frameworks*, 2000.

[7] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, *Simple Object Access Protocol (SOAP) 1.1.* World Wide Web Consortium, May 2000. `http://www.w3.org/TR/SOAP/`.

[8] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, pp. 38–47, Feb. 1996.

[9] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*, (Oakland, CA), pp. 164–173, IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 1996.

[10] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Communications of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[11] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. 10, pp. 265–310, Nov. 1992.

[12] J. Kohl and C. Neuman, "RFC 1510: The Kerberos Network Authentication Service (V5)," RFC 1510, The Internet Engineering Task Force, Sept. 1993.

[13] L. Gong, "A secure identity-based capability system," in *Proceedings of the IEEE Symposium on Security and Privacy*, (Los Angeles, CA), pp. 55–63, IEEE, IEEE Computer Society Press, May 1989.

[14] J. A. Bull, L. Gong, and K. R. Sollins, "Towards security in an open systems federation," in *European Symposium on Research in Computer Security (ESORICS)*, pp. 3–20, 1992.

[15] R. Hayton, *OASIS: An Open Architecture for Secure Interworking Services*. PhD thesis, Univeristy of Cambridge Computer Laboratory, June 1996. Technical Report No. 399.

[16] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The role of trust management in distributed systems security," in *Proceedings of Fourth International Workshop on Mobile Object Systems: Secure Internet Mobile Computations (MOS '98, Brussels, Belgium)*, no. 1603 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 185–210, Springer-Verlag, July 1999.

[17] R. L. Rivest and B. Lampson, "SDSI–A simple distributed security infrastructure." See `http://theory.lcs.mit.edu/~rivest/sdsi10.ps`, Aug. 1996.

[18] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI certificate theory," RFC 2693, Internet Engineering Task Force, Sept. 1999. See `http://www.ietf.org/rfc/rfc2693.txt`.

[19] M. Blaze, J. Feigenbaum, and A. D. Keromytis, "KeyNote: Trust management for public-key infrastructures," in *Security Protocols – 6th International Workshop* (B. Christianson, B. Crispo, W. S. Harbison, and M. Roe, eds.), no. 1550 in Lecture Notes in Computer Science, (Cambridge, United Kingdom), pp. 59–66, Springer-Verlag, Berlin Germany, Apr. 1999.

[20] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The KeyNote trust management system," Internet Request for Comment RFC 2704, Internet Engineering Task Force, Sept. 1999. Version 2.

[21] R. Sandhu, "Engineering authority and trust in cyberspace: The OM-AM and RBAC way," in *Proc. 5th ACM Workshop on Role-Based Access Control (RBAC-00)*, (New York, NY), pp. 111–119, ACM Press, July 26–27 2000.

[22] D. Bell and L. LaPadula, "Secure computer systems: Mathematical foundations," Tech. Rep. MTR-2547, Vol. I – III, MITRE Corporation, Bedford, MA, Nov. 1973.

[23] K. Biba, "Integrity consideration for secure computer systems," Tech. Rep. MTR-3153, MITRE Corporation, Bedford, MA, Apr. 1975.

[24] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *Proceedings of the 1987 IEEE Symposium on Security and Privacy (SSP '87)*, (Los Angeles, CA), pp. 184–195, IEEE Computer Society Press, Apr. 1987.

[25] D. F. C. Brewer and M. J. Nash, "The Chinese Wall security policy," in *Proc. IEEE Symposium on Security and Privacy*, pp. 206–214, 1989.

[26] D. Ferraiolo and R. Kuhn, "Role-based access controls," in *Proc. 15th NIST-NCSC National Computer Security Conference*, pp. 554–563, 1992.

[27] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, pp. 613–615, Oct. 1973.

[28] E. Amoroso, *Fundamentals of Computer Security Technology*. Prentice Hall, Apr. 1994. ISBN 0-13108-929-3.

[29] R. Sandhu, "Transaction control expressions for separation of duties," in *4th Aerospace Computer Security Conference*, pp. 282–286, Dec. 1988.

[30] R. S. Sandhu, "Separation of duties in computerized information systems," in *IFIP Workshop on Database Security*, pp. 179–190, 1990.

[31] R. T. Simon and M. E. Zurko, "Separation of duty in role-based environments," in *Proc. 10th IEEE Computer Security Foundations Workshop (Rockport, MA, June 10–12)*, (Los Alamitos, CA), pp. 183–194, IEEE Computer Society Press, June 1997.

[32] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo, "On the formal definition of separation-of-duty policies and their composition," in *1998 IEEE Symposium on Security and Privacy (SSP '98)*, (Washington - Brussels - Tokyo), pp. 172–185, IEEE Press, May 1998.

[33] B. Lampson, "Protection," in *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, (Princeton University), pp. 437–443, 1971.

[34] R. W. Baldwin, "Naming and grouping privileges to simplify security management in large database," in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland, CA)*, (Los Alamitos, CA), pp. 116–132, IEEE Computer Society Press, May 1990.

[35] S. A. Demurjian, M.-Y. Hu, T. C. Ting, and D. Kleinman, "Towards an authorization mechanism for user-role based security in an object-oriented design model," in *Proceedings of the 12th Annual International Phoenix Conference on Computers and Communications (Tempe, AR)* (J. Weeldreyer, ed.), (Los Alamitos, CA), pp. 195–202, IEEE Computer Society Press, Mar. 1993.

[36] M. Nyanchama and S. Osborn, "Role-based security: Pros, cons & some research directions," *ACM SIGSAC Review*, vol. 2, pp. 11–17, June 1993. ACM Press.

[37] M. Nyanchama and S. Osborn, "Access rights administration in role-based security systems," in *Proc. 8th IFIP WG 11.3 Working Conference on Database Security (Database Security VIII: Status and Prospects) (Bad Salzdetfurth, Germany, Aug. 23–26* (J. Biskup, M. Morgernstern, and C. Landwehr, eds.), vol. A-60 of *IFIP Transactions*, (Amsterdam, The Netherlands), North-Holland (Elsevier), 1995.

[38] M. Nyanchama and S. Osborn, "The role graph model and conflict of interest," *ACM Transactions on Information and System Security*, vol. 2, pp. 3–33, Feb. 1999. ACM Press, New York, NY.

[39] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn, "A role-based access control model and reference implementation within a corporate intranet," *ACM Transactions on Information and System Security*, vol. 2, pp. 34–64, Feb. 1999.

[40] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The NIST model for role-based access control: Towards a unified standard," in *Proc. 5th ACM Workshop on Role-Based Access Control (RBAC-00)*, (N.Y.), pp. 47–64, ACM Press, July 26–27 2000.

[41] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM Transactions on Information and System Security*, vol. 4, pp. 224–274, Aug. 2001.

[42] D. Ferraiolo, J. Cugini, and R. Kuhn, "Role based access control (RBAC): Features and motivations," in *Annual Computer Security Applications Conference*, IEEE Computer Society Press, 1995.

[43] D. F. Ferraiolo and J. Barkley, "Specifying and managing role-based access control within a corporate intranet," in *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC '97, Fairfax, VA, Nov. 6–7)*, (New York, NY), pp. 77–82, ACM Press, Nov. 6–7 1997.

[44] R. Sandhu, "Role activation hierarchies," in *Proc. 3rd ACM Workshop on Role-Based Access Control (Fairfax, VA, October 22-23)*, (New York, NY), pp. 33–40, ACM Press, Oct. 1998.

[45] J. D. Moffett, "Control principles and access right inheritance through role hierarchies," in *Proc. 3rd ACM Workshop on Role-Based Access Control (Fairfax, VA, October 22-23)*, (New York, NY), pp. 63–69, ACM Press, Oct. 1998.

[46] D. R. Kuhn, "Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems," in *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC '97, Fairfax, VA, Nov. 6–7)*, (New York, NY), pp. 23–30, ACM Press, Nov. 6–7 1997.

[47] L. Giuri and P. Iglio, "A formal model for role-based access control with constraints," in *Proc. 9th IEEE Computer Security Foundations Workshop*, (Los Alamitos, CA), pp. 136–145, IEEE Computer Society Press, 1996.

[48] R. Sandhu, V. Bhamidipati, and Q. Munawer, "The ARBAC97 model for role-based administration of roles," *ACM Transactions on Information and System Security*, vol. 2, pp. 105–135, Feb. 1999.

[49] S. Osborn, "Mandatory access control and role-based access control revisited," in *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC-97)*, (New York, NY), pp. 31–40, ACM Press, Nov. 6–7 1997.

[50] R. Sandhu and Q. Munawer, "How to do DAC using roles," in *Proceedings of the 3rd ACM Workshop on Role-Based Access Control (RBAC-98)*, (New York, NY), pp. 47–54, ACM Press, Oct. 22–23 1998.

[51] S. Osborn, R. Sandhu, and Q. Munawer, "Configuring role-based access control to enforce mandatory and discretionary access control policies," *ACM Transactions on Information and System Security*, vol. 3, pp. 85–106, May 2000.

[52] R. M. Needham and A. H. Herbert, *The Cambridge Distributed Computing System.* Addison Wesley, Jan. 1982. ISBN 0-20114-092-6.

[53] J. Bacon, I. Leslie, and R. Needham, "Distributed computing with a processor bank," Tech. Rep. 168, University of Cambridge Computer Laboratory, Apr. 1989.

[54] A. D. Birrell and R. M. Needham, "A universal file server," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 450–453, Sept. 1980.

[55] J. Dion, "The Cambridge file server," *ACM Operating Systems Review*, vol. 14, no. 4, pp. 26–35, 1980.

[56] W. A. Wulf, E. S. Cohen, W. M. Corwin, A. K. Jones, R. Levin, C. Pierson, and F. J. Pollack, "HYDRA: The kernel of a multiprocessor operating system," *Communications of the ACM*, vol. 17, pp. 337–345, June 1974.

[57] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum, "Experience with the amoeba distributed operating system," *Communications of the ACM*, vol. 33, pp. 46–63, Dec. 1990.

[58] S. J. Mullender, C. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Stavern, "Amoeba: a distributed operating system for the 1990s.," *IEEE Computer*, vol. 23, pp. 44–53, May 1990.

[59] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, "Kerberos authentication and authorization system," Project Athena Technical Plan, Section E.2.1, MIT Laboratory for Computer Science, Cambridge, MA, Dec. 1987.

[60] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *USENIX Conference Proceedings (Dallas, TX, USA)* (USENIX Association, ed.), (Berkeley, CA, USA), pp. 191–202, USENIX Association, Mar. 1988.

[61] D. E. Denning and M. S. Sacco, "Timestamps in key distribution protocols," *Communications of the ACM*, vol. 24, pp. 533–536, Aug. 1981.

[62] R. M. Needham and M. D. Schroeder, "Authentication revisited," *ACM Operating Systems Review*, vol. 21, p. 7, Jan. 1987.

[63] D. Otway and O. Rees, "Efficient and timely mutual authentication," *ACM Operating Systems Review*, vol. 21, pp. 8–10, Jan. 1987.

[64] S. M. Bellovin and M. Merritt, "Limitations of the Kerberos authentication system," *Computer Communication Review*, vol. 20, no. 5, pp. 119–132, 1990. ACM Press, New York, NY.

[65] The Open Group, *F201: DCE 1.2.2 Introduction to OSF DCE*, Nov. 1997. ISBN 1-85912-182-9.

[66] C. B. Neuman, "Proxy-based authorization and accounting for distributed systems," in *13th International Conference on Distributed Computing Systems*, pp. 283–291, May 1993.

[67] R. Hayton, J. Bacon, and K. Moody, "OASIS: Access control in an open, distributed environment," in *Proceedings of IEEE Symposium on Security and Privacy (Oakland, CA, May 3–6)*, (Los Alamitos, CA), IEEE Computer Society Press, 1998.

[68] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, Nov. 1976.

[69] P. R. Zimmermann, *The Official PGP User's Guide*. Cambridge, MA, USA: MIT Press, 1995.

[70] S. Garfinkel, *PGP: Pretty Good Privacy*. Sebastopol, CA: O'Reilly & Associates, Inc., 1995. ISBN 1-56592-098-8.

[71] CCITT (Consultative Committee on International Telegraphy and Telephony), *CCITT Recommendation X.509: The Directory – Authentication Framework*, 1988.

[72] ITU-T (Telecommunication Standardization Sector, International Telecommunication Union), Geneva, Switzerland, *ITU-T Recommendation X.509: The Directory – Authentication Framework*, 1997.

[73] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 public key infrastructure certificate and CRL profile," RFC 2459, Internet Engineering Task Force, Jan. 1999. See `http://www.ietf.org/rfc/rfc2459.txt`.

[74] ANSI (American National Standards Institute), Washington, DC, *ANSI X9.45: Enhanced Management Controls Using Digital Signatures and Attribute Certificates*, 1999.

[75] S. Farrell and R. Housley, "An Internet attribute certificate profile for authorization," Internet Draft draft-ietf-pkix-ac509prof-09, Internet Engineering Task Force, June 2001. See `http://www.ietf.org/internet-drafts/draft-ietf-pkix-ac509prof-09.txt`.

[76] C. M. Ellison, "SPKI requirements," RFC 2692, Internet Engineering Task Force Draft IETF, Sept. 1999. See `http://www.ietf.org/rfc/rfc2692.txt`.

[77] L. M. Kohnfelder, "Towards a practical public-key cryptosystem," B.Sc thesis, MIT Departement of Electrical Engineering, May 1978.

[78] ITU-T (Telecommunication Standardization Sector, International Telecommunication Union), Geneva, Switzerland, *ITU-T Recommendation X.509: The Directory – Authentication Framework*, 1993. (also ISO/IEC 9594-8, 1995).

[79] M. Myers, C. Adams, D. Solo, and D. Kemp, "Internet X.509 certificate request message format," RFC 2511, Internet Engineering Task Force, Mar. 1999. See `http://www.ietf.org/rfc/rfc2511.txt`.

[80] M. Myers, X. Liu, J. Schaad, and J. Weinstein, "Certificate management messages over CMS," RFC 2797, Internet Engineering Task Force, Apr. 2000. See `http://www.ietf.org/rfc/rfc2797.txt`.

[81] S. Boeyen, T. Howes, and P. Richard, "Internet X.509 public key infrastructure: Operational protocols - LDAPv2," RFC 2559, Internet Engineering Task Force, Apr. 1999. See `http://www.ietf.org/rfc/rfc2559.txt`.

[82] R. Housley and P. Hoffman, "Internet X.509 public key infrastructure: Operational protocols - FTP and HTTP," RFC 2585, Internet Engineering Task Force, May 1999. See `http://www.ietf.org/rfc/rfc2585.txt`.

[83] I. Lehti and P. Nikander, "Certifying trust," in *Proc. 1st International Public Key Cryptography Conference*, no. 1431 in Lecture Notes in Computer Science, pp. 83–98, 1998.

[84] J. Linn, "Trust models and management in public-key infrastructures," technical report, RSA Data Security, Inc., Redwood City, CA, USA, Nov. 2000.

[85] A. Jøsang, I. G. Pedersen, and D. Povey, "PKI seeks a trusting relationship," in *Proceedings of Fifth Australasian Conference on Information Security and Privacy (ACISP 2000, Brisbane, Australia)* (E. Dawson, A. Clark, and C. Boyd, eds.), no. 1841 in Lecture Notes in Computer Science, (Berlin, Germany), Springer-Verlag, July 2000.

[86] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, "X.509 Internet public key infrastructure: Online certificate status protocol - OCSP," RFC 2560, Internet Engineering Task Force, June 1999. See `http://www.ietf.org/rfc/rfc2560.txt`.

[87] W. Johnston, S. Mudumbai, and M. Thompson, "Authorization and attribute certificates for widely distributed access control," in *Proceedings of the 7th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '98, Stanford, CA)*, (Los Alamitos, CA), IEEE Computer Society Press, June 1998.

[88] M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari, "Certificate-based access control for widely distributed resources," in *Proceedings of the 8th USENIX Security Symposium (SECURITY-99)*, (Berkely, CA), pp. 215–228, Usenix Association, Aug. 23–26 1999.

[89] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, "A security architecture for computational grids," in *Proc. 5th ACM Conference on Computer and Communications Security (CCS-5, San Francisco, CA)*, (New York, NY), pp. 83–92, ACM Press, Nov. 1998.

[90] R. Butler, V. Welch, D. Engert, I. Foster, S. Tuecke, J. Volmer, and C. Kesselman, "A national-scale authentication infrastructure," *IEEE Computer*, vol. 33, pp. 60–66, Dec. 2000.

[91] D. Chadwick and A. Otenko, "RBAC policies in XML for X.509 based privilege management," in *Proceedings of the 17th International Conference on Information Security*, (Cairo, Egypt), May 2002.

[92] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "Simple public key certificate," Internet Draft draft-ietf-spki-cert-structure-06, Internet Engineering Task Force, Jan. 1999. See `http://world.std.com/~cme/spki.txt`.

[93] M. Blaze, J. Feigenbaum, and J. Lacy, "Managing trust in medical information systems," Tech. Rep. 96.14.1, AT&T, 1996.

[94] M. Blaze, J. Ioannidis, and A. Keromytis, "Trust management for IPsec," in *Proceedings of the Network and Distributed System Security Symposium: 2001 (NDSS'01, San Diego, CA)*, (Reston, Virginia), Internet Society, Feb. 2001.

[95] M. Blaze, J. Ioannidis, and A. D. Keromytis, "Trust management for IPsec," *ACM Transactions on Information and System Security*, vol. 5, no. 3, pp. 95–118, 2002.

[96] M. Blaze, J. Feigenbaum, P. Resnick, and M. Strauss, "Managing trust in an information-labeling system," *European Transactions on Telecommunications*, vol. 8, no. 5, pp. 491–501, 1997.

[97] M. Blaze, J. Feigenbaum, and M. Strauss, "Compliance checking in the policy maker trust management system," in *Proceedings of the Financial Cryptography 1998 (FC'98, Anguilla, British West Indies)*, no. 1465 in Lecture Notes in Computer Science, (Berlin, Germany), pp. 254–274, Springer-Verlag, Feb. 1998.

[98] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss, "REFEREE: Trust management for web applications," in *Proc. 6h International World-Wide Web Conference (WWW6, Santa Clara, CA)*, Apr. 1997.

[99] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss, "REFEREE: Trust management for web applications," *The World Wide Web Journal*, vol. 2, no. 3, pp. 127–139, 1997. Available at `http://www.w3j.com/`.

[100] T. Krauskopf, J. Miller, P. Resnick, and W. Treese, "PICS label distribution label syntax and communication protocols, version 1.1," Recommendation REC-PICS-labels-961031, World Wide Web Consortium, Oct. 1996. Available at `http://www.w3.org/TR/REC-PICS-labels`.

[101] Herzberg, Mass, Mihaeli, Naor, and Ravid, "Access control meets public key infrastructure, or: Assigning roles to strangers," in *RSP: 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[102] A. Herzberg and Y. Mass, "Relying party credentials framework," in *Proc. RSA Conference 2001*, vol. 2020 of *Lecture Notes in Computer Science*, (Heidelberg, Germany), pp. 328–343, Springer-Verlag, Apr. 2001.

[103] N. Li, J. C. Mitchell, and W. H. Winsborough, "Design of a role-based trust-management framework," in *IEEE Symposium on Security and Privacy*, (Los Angeles, CA), pp. 114–130, IEEE Computer Society Press, May 2002.

[104] U.S. Department of Defense, *DoD 5200.28-STD: Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC)*, 1985.

[105] B. J. Fogg and H. Tseng, "The elements of computer credibility," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI-99)*, (New York), pp. 80–87, ACM Press, May 15–20 1999.

[106] S. Tseng and B. J. Fogg, "Credibility and computing technology," *Communications of the ACM*, vol. 42, pp. 39–44, May 1999.

[107] R. Yahalom, B. Klein, and T. Beth, "Trust relationships in secure systems-A distributed authentication perspective," in *Proceedings of the 1993 IEEE Computer Society Symposium on Security and Privacy (SSP '93)*, (Washington - Brussels - Tokyo), pp. 150–164, IEEE, May 1993.

[108] A. Jøsang, "Prospectives for modelling trust in information security," in *Proc. 2nd Information Security and Privacy Conference – ACISP '97*, pp. 2–13, 1997.

[109] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication," *ACM Transactions on Computer Systems*, vol. 8, pp. 18–36, Feb. 1990.

[110] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A calculus for access control in distributed systems," *ACM Transactions on Programming Languages and Systems*, vol. 15, pp. 706–734, Sept. 1993.

[111] B. Crispo, "Delegation of responsibilities," in *Proc. 6th International Security Protocols Workshop (Cambridge, UK, April 15–17)*, no. 1550 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 118–130, Springer-Verlag, 1998.

[112] A. Nash, B. Duane, D. Brink, and C. Joseph, *PKI: Implementing and Managing E-Security.* McGraw-Hill Professional Publishing, Mar. 2001. ISBN 0-0721-3123-3.

[113] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Key management involving multiple domains*, ch. 13.6, pp. 570–577. CRC Press, 1997.

[114] A. Jøsang, "The right type of trust for distributed systems," in *Proceedings of ACM Workshop on New Security Paradigms*, ACM SIGSAC, ACM Press, Sept. 1996.

[115] B. Shneiderman, "Designing trust into online experiences," *Communications of the ACM*, vol. 43, pp. 57–59, Dec. 2000.

[116] D. W. Manchala, "E-commerce trust metrics and models," *IEEE Internet Computing*, vol. 4, no. 2, pp. 36–44, 2000.

[117] T. Grandison and M. Sloman, "A survey of trust in Internet applications," *IEEE Communications Surveys & Tutorials*, vol. 3, no. 4, 2000.

[118] S. Einwiller, "Analyzing the potential of the key dimensions of reputation to create trust in electronic commerce," in *Proc. 8th Research Symposium on Emerging Electronic Markets (RSEEM'01, Maastricht, The Netherlands)*, Sept. 2001.

[119] A. Abdul-Rahman and S. Hailes, "Using recommendations for managing trust in distributed systems," in *Proc. IEEE Malaysia International Conference on Communication '97 (MICC'97)*, (Kuala Lumpur, Malaysia), Nov. 1997.

[120] A. Abdul-Rahman and S. Hailes, "A distributed trust model," in *Proceedings of the ACM Workshop on New Security Paradigms*, (Cumbria, United Kingdom), pp. 48–60, ACM SIGSAC, ACM Press, Sept. 1997.

[121] P. Resnick, R. Zeckhauser, E. Friedman, and K. Kuwabara, "Reputation systems," *Communications of the ACM*, vol. 43, pp. 45–48, Dec. 2000.

[122] J. S. Olson and G. M. Olson, "i2i trust in e-commerce," *Communications of the ACM*, vol. 43, pp. 41–44, Dec. 2000.

[123] A. Abdul-Rahman and S. Hailes, "Supporting trust in virtual communities," in *Proc. 33th Hawaii International Conference on System Sciences*, IEEE Press, January 2000.

[124] F. Fukuyama, *Trust : The Social Virtues and the Creation of Prosperity.* New York, NY: Free Press, June 1996. ISBN 0684825252.

[125] D. Gambetta, "Can we trust trust?," in *Trust: Making and Breaking Cooperative Relations* (D. Gambetta, ed.), ch. 13, pp. 213–237, New York, NY: Basil Blackwell, 1988.

[126] B. Christianson and W. S. Harbison, "Why isn't trust transitive?," in *Proc. 4th International Security Protocols Conference*, pp. 171–176, 1996.

[127] N. Luhmann, *Trust and Power.* New York, NY: Wiley, 1979.

[128] B. A. Misztal, *Trust in Modern Societies : The Search for the Bases of Social Order.* Cambridge, MA: Polity Press, 1996. ISBN 0745612482.

[129] D. Fahrenholtz and A. Bartelt, "Towards a sociological view of trust in computer science," in *Proc. 8th Research Symposium on Emerging Electronic Markets (RSEEM'01, Maastricht, The Netherlands)*, Sept. 2001.

[130] D. Schoder and P.-L. Yin, "Building firm trust online," *Communications of the ACM*, vol. 43, pp. 73–79, Dec. 2000.

[131] B. Schneier, "A primer on authentication and digital signatures," *Computer Security Journal*, vol. 10, no. 2, pp. 38–40, 1994.

[132] S. Micali, "Enhanced certificate revocation system." Technical memo MIT/LCS/TM-542, 1995. Available at `ftp://ftp-pubs.lcs.mit.edu/pub/lcs-pubs/tm.outbox/MIT-LCS-TM-542.ps.gz`.

[133] M. Naor and K. Nissim, "Certificate revocation and certificate update," in *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, (Berkeley), pp. 217–228, Usenix Association, Jan. 26–29 1998.

[134] W. Aiello, S. Lodha, and R. Ostrovsky, "Fast digital identity revocation (extended abstract)," in *18th Annual International Cryptology Conference (CRYPTO'98, Santa Barbara, CA)*, no. 1462 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 137–152, Springer-Verlag, Aug. 1998.

[135] I. Gassko, P. Gemmell, and P. D. MacKenzie, "Efficient and fresh cerification," in *Proc. 3rd International Workshop on Practice and Theory in Public Key Cryptography (PKC 2000, Melbourne, Australia)*, no. 1751 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 342–353, Springer-Verlag, Jan. 2000.

[136] R. Wright, P. D. Lincoln, and J. K. Millen, "Efficient fault-tolerant certificate revocation," in *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS-00)*, (New York, NY), pp. 19–24, ACM Press, Nov. 1–4 2000.

[137] A. Jøsang, "A subjective metric of authentication," in *Proc. 5th European Symposium on Research in Computer Security (ESORICS'98, Louvain-la-Neuve, Belgium)*, no. 1485 in Lecture Notes in Computer Science, (Heidelberg, Germany), Springer-Verlag, 1998.

[138] A. Jøsang and S. J. Knapskog, "A metric for trusted systems," in *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pp. 16–29, 1998.

[139] D. W. Manchala, "Trust metrics, models and protocols for electronic commerce transactions," in *Proc. 18th International Conference on Distributed Computing Systems (ICDCS'98)*, (Amsterdam, The Netherlands), pp. 312–321, IEEE, May 1998.

[140] S. P. Marsh, *Formalising Trust as a Computational Concept.* PhD thesis, University of Stirling, Apr. 1994.

[141] T. Beth, M. Borcherding, and B. Klein, "Valuation of trust in open networks," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS '94, Brighton, UK)*, no. 875 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 3–18, Springer-Verlag, Nov. 1994.

[142] U. Maurer, "Modelling a public-key infrastructure," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS '96, Rome, Italy)*, no. 1146 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 325–350, Springer-Verlag, Sept. 1996.

[143] C. M. Ellison, "Naming and certificates," in *Proceedings of the tenth conference on Computers, freedom and privacy: challenging the assumptions*, (Toronto, Canada), pp. 213–217, April 2000.

[144] T. Aura, "Distributed access-rights managements with delegations certificates," in *Proceedings of Fourth International Workshop on Mobile Object Systems: Secure Internet Mobile Computations (MOS '98, Brussels, Belgium)*, no. 1603 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 211–235, Springer-Verlag, July 1999.

[145] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, eds., *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, Jan. 2000. ISBN 1-55860-647-5.

[146] W. Rubin and M. Brain, *Understanding DCOM.* Englewood Cliffs, NJ: Prentice-Hall, 1999. Includes CD-ROM.

[147] ISO (International Organization for Standardization), Geneva, Switzerland, *ISO 8601-2000: Representations of dates and times, 2000-12-21*, 2000.

[148] B. Harbison, "Delegating trust (transcript of discussion)," in *Proc. 6th International Security Protocols Workshop (Cambridge, UK, April 15–17)*, no. 1550 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 108–117, Springer-Verlag, 1998.

[149] World Wide Web Consortium, *Extensible Markup Language (XML) 1.0*, 2nd ed., Oct. 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[150] Organization for the Advancement of Structured Information Standards (OASIS), *ebXML Technical Architecture Specification*, v 1.0.4 ed., Feb. 2001. http://www.ebxml.org/specs/ebTA.pdf.

[151] Organization for the Advancement of Structured Information Standards (OASIS), *ebXML Business Process Specification Schema*, v 1.01 ed., May 2001. http://www.ebxml.org/specs/ebBPSS.pdf.

[152] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, and P. H.-B. et. al, *Web Services Security (WS-Security) Version 1.0*, Apr. 2002. http://www-106.ibm.com/developerworks/library/ws-secure/.

[153] W. Ford, P. Hallam-Baker, B. Fox, B. Dillaway, B. LaMacchia, J. Epstein, and J. Lapp, *XML Key Management Specification (XKMS) (W3C Note)*. World Wide Web Consortium, Mar. 2001. http://www.w3.org/TR/xkms/.

[154] Organization for the Advancement of Structured Information Standards (OASIS), *Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML)*, May 2002. http://www.oasis-open.org/committees/security/docs/cs-sstc-core-01.pdf.

[155] Organization for the Advancement of Structured Information Standards (OASIS), *Bindings and Profiles for the OASIS Security Assertion Markup Language (SAML)*, May 2002. http://www.oasis-open.org/committees/security/docs/cs-sstc-bindings-01.pdf.

[156] M. Gudgin, M. Hadley, J.-J. Moreau, and H. F. Nielsen, *SOAP Version 1.2 Part 1: Messaging Framework (W3C Working Draft 17 December 2001)*. World Wide Web Consortium, Dec. 2001. http://www.w3.org/TR/soap12-part1/.

[157] M. Gudgin, M. Hadley, J.-J. Moreau, and H. F. Nielsen, *SOAP Version 1.2 Part 2: Adjuncts (W3C Working Draft 17 December 2001)*. World Wide Web Consortium, Dec. 2001. http://www.w3.org/TR/soap12-part2/.

[158] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1 (W3C Note 15 March 2001)*. World Wide Web Consortium, Mar. 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[159] World Wide Web Consortium, *XML Schema Part 1: Structures (W3C Recommendation 2 May 2001*, May 2001. http://www.w3.org/TR/xmlschema-1/.

[160] World Wide Web Consortium, *XML Schema Part 2: Datatypes (W3C Recommendation 2 May 2001*, May 2001. http://www.w3.org/TR/xmlschema-2/.

[161] UDDI.org, *UDDI Version 2.0: API Specification (UDDI Open Draft Specification 8 June 2001*, June 2001. http://www.uddi.org/pubs/ProgrammersAPI-V2.00-Open-20010608.pdf.

[162] UDDI.org, *UDDI Version 2.0: Data Structure Reference (UDDI Open Draft Specification 8 June 2001*, June 2001. http://www.uddi.org/pubs/DataStructure-V2.00-Open-20010608.pdf.

[163] T. Howes and M. Smith, "An LDAP URL format," Internet Request for Comment RFC 1959, Internet Engineering Task Force, June 1996.

[164] T. Dierks and C. Allen, "The TLS protocol version 1.0," RFC 2246, Internet Engineering Task Force, Jan. 1999. Proposed Standard.

[165] World Wide Web Consortium, *Namespaces in XML*, Jan. 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114/.

[166] World Wide Web Consortium, *XML-Signature Syntax and Processing (W3C Recommendation)*, Feb. 2002. http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/.

[167] World Wide Web Consortium, *XML Path Language (XPath) 2.0 (W3C Working Draft)*, Apr. 2002. http://www.w3.org/TR/xpath20/.

[168] W. Winsborough, K. Seamons, and V. Jones, "Automated trust negotiation," Tech. Rep. TR-2000-05, Department of Computer Science, North Carolina State University, Apr. 24 2000. Mon, 24 Apr 2000 17:07:47 GMT.

[169] K. E. Seamons, M. Winslett, and T. Yu, "Limiting the disclosure of access control policies during automated trust negotiation," in *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2001, San Diego, CA)*, (San Diego, California), Internet Society, Feb. 2001.

[170] W. H. Winsborough and N. Li, "Towards practical trust negotiation," in *Proc. 3rd Workshop on Policies for Distributed Systems and Networks (Policy 2002)*, pp. 92–103, June 2002.

[171] Apache Software Foundation, http://httpd.apache.org/, *Apache HTTP Server Project*, 1999.

[172] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," RFC 2616, The Internet Society, June 1999. See http://www.ietf.org/rfc/rfc2616.txt.

[173] H. Nielsen, P. Leach, and S. Lawrence, "An HTTP extension framework," RFC 2774, The Internet Society, Feb. 2000. See http://www.ietf.org/rfc/rfc2774.txt.

[174] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "HTTP authentication: Basic and digest access authentication," RFC 2617, The Internet Society, June 1999. See http://www.ietf.org/rfc/rfc2617.txt.

[175] N. Freed and N. Borenstein, "Multipurpose internet mail extensions (MIME) part two: Media types," rfc, Internet Engineering Task Force Draft IETF, Nov. 1996. See http://www.ietf.org/rfc/rfc2046.txt.

[176] Microsoft Corp., http://www.passport.com/, *Microsoft .NET Passport*, 2002.

[177] Entrust Inc., http://www.entrust.com/getaccess, *Entrust GetAccess*, 2002.

[178] RSA Security Inc., http://www.rsasecurity.com/products/ClearTrust/index.html, *RSA ClearTrust*, 2002.

[179] *PHP4: Hypertext Preprocessor.* http://www.php.net/, 2001.

[180] *PayPal Internet Payment System.* www.paypal.com.

[181] T. Yu, M. Winslett, and K. E. Seamons, "Interoperable strategies in automated trust negotiation," in *Proceedings of the 8th ACM Conference on Computer and Communications Security (Philadelphia, PA, USA)* (P. Samarati, ed.), (New York, NY), pp. 146–155, ACM Press, Nov. 2001.

[182] World Wide Web Consortium, *XML Encryption Syntax and Processing W3C Candidate Recommendation*, Mar. 2002. http://www.w3.org/TR/xmlenc-core/.

[183] S. A. Brands, *Rethinking Public Key Infrastructures and Digital Certificates; Building in Privacy.* Cambridge, MA: MIT Press, Aug. 2000. ISBN 0-262-02491-8.