OCT 2 9 1997

Trust Management for the World Wide Web

by

Yang-hua Chu

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

June 13, 1997

Copyright 1997 Yang-hua Chu. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce
distribute publicly paper and electronic copies of this thesis
and to grant others the rights to do so.

Author _____
Department of Electrical Engineering and Computer Science

Certified by _____
Dr. Joan Feigenbaum
Technology Consultant, AT&T Labs—Research

Certified by _____
Lecturer

Accepted by _____
Professor Arthur C. Smith
Chairman, Department Committee on Graduate Thesis

PAN EX1005
Palo Alto Networks v. BT Americas
IPR2023-00889

Trust Management for the World Wide Web

by

Yang-hua Chu

Submitted to the Department of Electrical Engineering and Computer Science

June 13, 1997

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

ABSTRACT

Digital signatures alone are not sufficient for code signing and other Web applications: Signatures can solve the problems of message integrity and authentication, but they do not adequately address more general notions of security and trust. These applications require not only cryptographic tools for determining authenticity and message integrity but also a robust notion of "security policy" and a way to decide whether a request for action complies with a policy. For example, in a code-signing application, a user's security policy must state the properties that the code is required to have in order to be considered "safe" in the user's environment. Similarly, the entity signing the code must state precisely what properties he claims the code has.

My thesis will identify what trust management is in the context of the World Wide Web and propose a general architecture to close the gap between trust and cryptography. I will describe two specific languages for describing trust policies and a general mechanism for evaluating whether a request for action complies with policy.

Thesis Supervisor	Title	Affiliation
Dr. Joan Feigenbaum	Technology Consultant	AT&T Labs—Research
Dr. James S. Miller	Technology and Society Domain Leader	The World Wide Web Consortium, MIT Laboratory for Computer Science

ACKNOWLEDGEMENTS

First I thank my thesis supervisors, Dr. Joan Feigenbaum and Dr. Jim Miller. They were always ready to give me guidance and support when I encountered problems during my research and thesis writing. They also provided me invaluable opportunities to attend conferences and give presentations.

I was grateful to work with several talented researchers at AT&T Labs—Research, including Brian LaMacchia, Paul Resnick, and Martin Strauss. We co-developed REFEREE, which ultimately became the focus of my research and thesis work. Their enthusiasm and devotion to doing research made them inspiring role models.

Many thanks to the team members at the World Wide Web Consortium, where I spent the past year writing my thesis. Special thanks to the T&S team members Eui-suk Chung, Philip DesAutels, Rohit Khare, and Joseph Reagle. Their presence and encouragement make my daily work on the third floor of LCS fun and worthwhile. Special thanks to Philip, whom I spent a great deal of time with in the Digital Signature Initiative project, and Joseph, who lent me the thesis template.

Finally I have to thank my personal support team: my mom and dad, my brothers Yung-hua, Ching-hua, and Hao-hua, and my girlfriend Wendy. Although land and sea separated us most of the time, we were always connected deep in our hearts. Every bit of caring and encouragement was my most precious source of energy. There are no words that can express my gratitude to them.

Table of Contents

1 INTRODUCTION 9

2 TRUST MANAGEMENT 11

2.1 WHAT IS TRUST MANAGEMENT 11
2.2 TRUST MANAGEMENT INFRASTRUCTURE 12
2.3 REVIEW OF EXISTING TRUST SYSTEMS AND PROTOCOLS 15
2.3.1 PICS 15
2.3.2 X.509 16
2.3.3 PolicyMaker 18
2.3.4 Microsoft Authenticode 19
2.4 EXAMPLES OF TRUST MANAGEMENT PROBLEMS IN THE WWW 23
2.4.1 Code Distribution 23
2.4.2 Document Authentication 25

3 EXECUTION ENVIRONMENT 28

3.1 DESIGN GOAL 28
3.2 REFEREE 29
3.3 REFEREE INTERNAL ARCHITECTURE 30
3.4 REFEREE PRIMITIVE DATA TYPES 31
3.4.1 Tri-Value 31
3.4.2 Statement and Statement-List 32
3.4.3 Module Databases 32
3.5 BOOTSTRAPPING REFEREE 33
3.6 QUERYING REFEREE 34

4 POLICY LANGUAGE 35

4.1 DESIGN GOALS 35
4.2 PICSRULZ 36
4.3 PROFILES-0.92 38
4.4 SAMPLE POLICIES 43
4.4.1 Sample policy 1: determine Access Based on the URL 44
4.4.2 Sample policy 2: determine access based on PICS labels 44
4.4.3 Sample Policy 3: Determine Access Based on Multiple PICS Labels and Sources 46
4.4.4 Sample Policy 4: Defer Trust Using Extension Mechanism 47

5 REFEREE REFERENCE IMPLEMENTATION 49

5.1 JIGSAW PROXY: THE HOST APPLICATION 49
5.2 REFEREE IN THE JIGSAW PROXY 51
5.3 THE SCOPE OF THE REFEREE IMPLEMENTATION 51
5.4 AN EXECUTION TRACE 53
5.5 DISCUSSIONS 56

6 CONCLUSION 58

APPENDICES 59

APPENDIX A. MODIFIED BNF FOR PICSRULZ POLICY LANGUAGE 59
APPENDIX B. MODIFIED BNF FOR PROFILES-0.92 POLICY LANGUAGE 60
APPENDIX C. MODIFIED BNF FOR THE RETURNED STATEMENT-LIST OF LABEL LOADER 61

REFERENCES 62

List of Figures and Tables

FIGURE 2 DEPENDENCY GRAPH OF TRUST MANAGEMENT INFRASTRUCTURE COMPONENTS 14
FIGURE 3 PICS IN THE TRUST MANAGEMENT INFRASTRUCTURE 15
FIGURE 4 X.509 IN THE TRUST MANAGEMENT INFRASTRUCTURE 17
FIGURE 5 POLICYMAKER IN THE TRUST MANAGEMENT INFRASTRUCTURE 18
FIGURE 6 AUTHENTICODE IN THE TRUST MANAGEMENT INFRASTRUCTURE 20
FIGURE 7 AUTHENTICODE USER PERMISSION INTERFACE 21
FIGURE 8 CONFIGURING A LIST OF TRUSTED ENTITIES IN AUTHENTICODE 22
FIGURE 9 COOL GAME DOWNLOAD 23
FIGURE 10 A SNAPSHOT OF THE BOSTON GLOBE WEB DOCUMENT 25
FIGURE 11 FLOW CHART FOR SIGNING AND VERIFYING A DIGITAL SIGNATURE 26
FIGURE 12 REFEREE EXTERNAL API 29
FIGURE 13 SAMPLE BLOCK DIAGRAM OF REFEREE INTERNAL STRUCTURE 30
FIGURE 14 REQUIRED INTERFACE FOR EVERY REFEREE MODULE 31
FIGURE 15 SAMPLE REFEREE IMPLEMENTATION 34
FIGURE 16 JIGSAW PROXY ARCHITECTURE 50
FIGURE 17 SAMPLE REFEREE IMPLEMENTATION 54

TABLE 1 A SAMPLE MODULE DATABASE 33
TABLE 2 TRUTH TABLE FOR THE *AND* OPERATOR 41
TABLE 3 TRUTH TABLE FOR THE *OR* OPERATOR 41
TABLE 4 TRUTH TABLE FOR THE *NOT* OPERATOR 42
TABLE 5 TRUTH TABLE FOR THE TRUE-IF-UNKNOWN OPERATOR 42
TABLE 6 TRUTH TABLE FOR THE FALSE-IF-UNKNOWN OPERATOR 42

1 Introduction

Many activities of growing importance in the "information infrastructure," including electronic commerce and mobile programming, depend critically on precise and reliable ways to manage *trust*. Users will need to know how trustworthy information is before they act on it. For example, they will need to know where the information comes from (authentication), what kind of information it is (content), what it can do (capability), and whether it was altered during transmission (integrity). Without knowledge of what or whom to trust, users may treat a piece of potentially valuable information as yet another stream of random bits. Worse yet, malicious parties may lure users into believing that a false piece of information is trustworthy.

Many existing mechanisms and protocols address specific aspects of trust in the information infrastructure, but none provides a complete solution. For example, *digital signatures* allow publishers to create and distribute non-refutable proofs of authorship of documents. *Public key infrastructures* bind public keys to entities so that users can establish trust chains from digital signatures to signers. *Metadata formats* allow creators of information resources or trusted third parties to make assertions about these resources. Users can query and process the trusted assertions before deciding what to do with the information resources. Each of these mechanisms and protocols defines a subset of all potential trust problems and solves or partially solves this subset.

The goal of my research is to design a complete *trust management infrastructure*, in which trust is specified, disseminated, and evaluated in parallel with the information infrastructure. I have identified four major components of a trust management infrastructure: *the metadata format*, *the trust protocol*, *the trust policy language*, and *the execution environment*, which are defined in Chapter two. Under this framework of study, I discovered that most existing approaches to trust deal with metadata formats and trust protocols but lacked general trust policy languages for specifying user preferences and generic environments for evaluating them. This finding leads to my interest and involvement in REFEREE.

REFEREE is a result of collaboration among researchers from AT&T and W3C, including myself. It was designed to be a general-purpose execution environment for all Web applications requiring trust. REFEREE evaluates user policies in response to a host application's request for actions. Policies are treated as programs in REFEREE. For a given request, REFEREE invokes the appropriate user policy and interpreter module and returns to the host application an answer (with justification) to the question of whether or not the request complies with the policy.

The underlying architecture of REFEREE allows different trust policy languages and trust protocols to co-exist in one execution environment. They are treated as add-on software modules and can be installed or de-installed modularly. At the time of development, we were unable to find a suitable policy language to demonstrate all the features of REFEREE, and so we designed the Profiles-0.92 language.

In order to develop a deeper understanding of REFEREE and to demonstrate its feasibility, power, and efficiency, I built a reference implementation of the REFEREE

trust management system. The implementation includes a set of the core REFEREE data types and methods, a PICS protocol, and a Profiles-0.92 policy interpreter to evaluate polices based on the PICS metadata format. In addition, I implemented another policy language called PicsRULZ and integrated it into the reference implementation, in order to demonstrate REFEREE's ability to handle multiple policy languages in particular and multiple software modules generally.

This thesis is about the work I have done on trust management during the last year. Chapter two introduces readers to the term *trust management infrastructure* and explains how existing systems and protocols map into my framework of infrastructure. Chapter two also identifies trust management problems that are common to several current Web applications.

Chapter three is devoted to the REFEREE execution environment. It explains in detail its requirements, architectural design, primitive data types, and standard methods of bootstrapping and querying.

Chapter four describes two different policy languages, PicsRULZ and Profiles-0.92. They represent two different approaches to writing user policies. The chapter also provides four sample policies of varying degrees of complexity and typicality. These policies are expressed in both PicsRULZ and Profiles-0.92, so that I can compare and contrast the strengths and weaknesses of the two languages.

Chapter five describes my implementation work on REFEREE and analyzes the system from the implementation perspective. I chose Jigsaw proxy as the host application and Java Virtual Machine as the underlying REFEREE execution environment. The work sheds light on how to use REFEREE in a real-world application.

Chapter six concludes my thesis.

2 Trust Management

The term *trust management* has received a great deal of attention in the network security community since it was first introduced in the paper "Decentralized Trust Management" by Blaze, Feigenbaum, and Lacy [BFL96]. Many existing systems have since been identified as trust management systems in the sense of [BFL96], including PolicyMaker [BFL96], SDSI [RL96], SPKI [EFRT97], and X.509 [CCITT88a, CCITT88b]. People have compared and contrasted these systems and their capabilities and limitations.

This chapter reviews the concept of "trust management" as the starting point for my thesis work. Later discussions of REFEREE in Chapter three and PicsRULZ and Profiles-0.92 in Chapter four address specific components of "trust management".

Section one introduces the *trust management problem* in the [BFL96]. Section two presents my alternative notion of *trust management infrastructure*. Section three analyzes several well-known systems in the "trust management infrastructure" framework and highlights their strengths and weaknesses. Section four sets the context of my thesis work by identifying several common Web applications that have similar trust management needs.

2.1 What is Trust Management

As formulated by Blaze, Feigenbaum, and Lacy, trust management addresses the question "is this request, supported by these credentials, in compliance with this user policy?" The [BFL96] paper identified three components of trust management:

- security policies
- security credentials
- trust relationships

Security policies are local policies that an application trusts unconditionally. Security credentials are assertions about objects by trusted third parties. Trust relationships are special cases of security policies. An example in the paper illustrated the use and the interactions among the three components:

```
An electronic banking system must enable a bank to state that at
least k bank officers are needed to approve loans of $1,000,000 or
less (a policy), it must enable a bank employee to prove that he can
be counted as 1 out of k approvers (a credential), and it must
enable the bank to specify who may issue such credentials (a trust
relationship).
```

The paper referred to the study of the three components and their interactions as the *trust management problem*. The authors believe that the trust management problem is a distinct and an important aspect of security in network services and that such problems can be solved using a general mechanism that is independent of any particular application or service. They propose is a trust management layer that applications and services can build on top of.

PolicyMaker, described in [BFL96], is a trust management system designed to meet the needs of this layer. It is a three-part solution: a credential format to represent

authorization assertions, a security policy language to express user preferences, and an execution environment to evaluate certificates and policies. PolicyMaker broke new ground by expressing credentials and policies as programs. The execution environment acts like a database query engine: The host application sends to the execution environment a request for action and a user policy, and the environment returns an answer to the question of whether the credentials prove that the request complies with the policy.

What is missing from PolicyMaker is consideration of "trust protocols", in particular, of mechanisms for acquiring additional trust information in the course of evaluating policies. PolicyMaker assumes the application is responsible for providing all credentials at the time a query is made to the trust management engine. In practice, the "right" set of credentials is often determined by the semantics of the policy and the state of the evaluation. For example, the "right" set of credentials to validate a PICS label may depend on the type of signature and certificate, the intended use of the Web document, the semantics of the label, the processor speed, or the network connection. These factors are often known only at the time of policy evaluation. A better approach is to put trust protocols under policy control. That is, a policy is capable of determining how, where, when, and under what circumstances to invoke a trust protocol to fetch credentials.

Adding "trust protocols" to the execution environment necessitates substantial changes in the [BFL96] framework. First, a trust policy needs a language construct with which to invoke trust protocols. The PolicyMaker language is only able to express authorizations. Moreover, a trust policy needs to be able to parse the retrieved credentials in order to make intelligent trust decisions. PolicyMaker avoids this by requiring the host application to translate the credentials into a special format before querying the engine. Furthermore, the underlying execution environment needs to be powerful enough to handle protocol invocations during the interpretation of a trust policy. It needs either to run the protocol inside its environment or to delegate the request to another software module. Finally, a trust management system should be extensible enough to install and de-install trust protocols. A given trust policy does not know *a priori* which trust protocols are available. When a protocol is unknown during the policy interpretation, the execution environment should be able to install it.

The need to treat metadata formats, trust protocols, trust policy languages, and execution environments as distinct components in trust management is the main conceptual contribution of my work on *trust management infrastructure* as explained in the next section. My framework draws the lines that separate components and assigns duties to each of them. Later the readers will see the importance of this component-wise view of trust management. It leads to several important design decisions in the REFEREE architecture: Policies and protocols are both software components of REFEREE, and these components not only coexist under one execution environment but also work together by invoking each other through a standard REFEREE interface.

2.2 Trust Management Infrastructure

A trust management infrastructure is a conceptual framework for the design of a coherent solution to various trust decisions that must be made in what is commonly referred to as

the "information infrastructure". A trust management infrastructure allows parties to make trusted assertions about objects in the information infrastructure, and applications to acquire these assertions and make trust decisions based on them. The framework is independent of the trust criteria imposed by any particular application and of the type of assertions made by a trusted party. There are four components in a trust management infrastructure:

Metadata format

is a format for describing information about an object, often called an assertion system. Metadata exists in various forms and under various names, but its function is similar throughout. For example, the content-filtering community calls its metadata "labels"; an instance of a label is "the code pointed to by this URL is safe to download." The security community calls its metadata "identify certificates"; an example of an identify certificate is "this person is over 18 years of age." Metadata is the medium in which trust flows from the entity creating the metadata to the application making the trust decision. It represents a token of trust.

Metadata itself is an object; it may be described by other metadata. The ability to form a chain of metadata allows trust to branch and decentralize into a Web-like structure. Actually, a collection of interconnected metadata look just like objects in an information infrastructure without trust. It is the duty of the trust policies and trust protocols to weave these objects together and provide them meanings.

Trust protocol

is a method for applications to acquire assertions from third parties. In X.509, there is an algorithm to walk up the certification path (a hierarchical chain) and gather the appropriate certificates along the path for a given directory name. In PICS and PGP, there is no algorithm for finding the "right certificates"; some metadata come with the original source, and others are acquired from named trusted parties.

The sole duty of a trust protocol is to gather an appropriate set of trusted assertions in order for a given request to comply with a trust policy. Algorithms used in a trust protocol do not perform any trust evaluation; rather they help a trust policy to collect a set of metadata likely to be used during a policy evaluation.

Trust policy language

is a language to specify a set of criteria for an object to be trustworthy to perform a given action. For example, there can be a policy about "downloading and running Java Applets in my browser", which requires "signed credentials from two trusted parties asserting that the Applet contains no virus".

Chapter 4 is devoted to the discussion of trust policy languages. PicsRULZ and Profiles-0.92, described in that chapter, are examples of trust policy languages.

Execution environment

is an environment for interpreting trust policies and administering trust protocols. An execution environment takes requests from its host application and returns an answer that is compliant with trust policies.

Chapter 3 is devoted to the discussion of execution environments. Both PolicyMaker and REFEREE have a notion of a general-purpose execution environment in which policies are evaluated.

Instances of the components work together to form a *trust management system*, which is an instance of a trust management infrastructure. For example, PolicyMaker trust management system has a metadata format, a trust policy language, and an execution environment. How components interact with each other may be application specific. This section provides a rule of thumb on how components *should* interact, based on their defined properties. Figure 2 shows a component dependency graph in the trust management infrastructure. Diamonds represent components in the trust management infrastructure and arrows represent dependency relations.

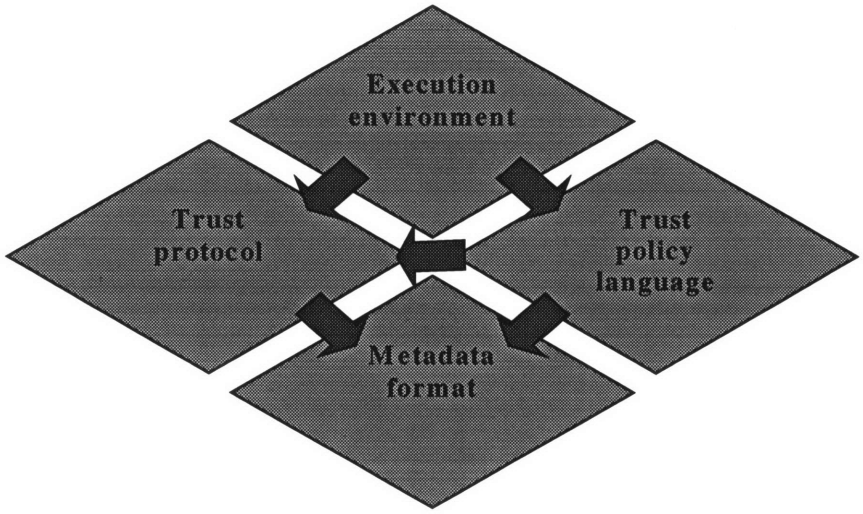

Figure 2 Dependency Graph of Trust Management Infrastructure Components

A metadata format is independent of any other components in the trust management infrastructure. It can be distributed by multiple protocols and operated on by multiple trust policy languages. For example, it is possible to represent SDSI certificates as PICS labels and use them in the SDSI public-key distribution protocol.

A trust protocol generally depends on the metadata format. A typical protocol contains methods to query specific metadata, heuristics to chain the metadata together, and ways to transport them. The specific properties in the metadata format enable the trust protocol to perform these methods.

A trust policy language depends on both the trust protocol and the metadata format. It must understand the syntax and the semantics of the metadata in order to write a policy on it. A trust policy may depend on trust protocols in order to fetch metadata at runtime.

An execution environment depends on both the trust policy language and the trust protocol. The underlying execution environment in the system needs to be powerful enough to interpret the policies and run the protocols. However, an execution environment need not understand the syntax or the semantics of the metadata formats directly. It is the duty of trust policies to parse the syntax and reason about the semantics.

Most existing trust systems can be mapped into this infrastructure, marked by their component dependencies. Not surprising, these solutions are geared toward neither a single component nor a complete infrastructure. For example, PICS is both a metadata format and a trust protocol, but it has neither a policy language to express trust relationships nor an execution environment to evaluate it. The next section discusses some well-known trust systems in greater detail and fits them into this framework.

2.3 Review of Existing Trust Systems and Protocols

There are many existing systems and protocols built to deal with trust issues; none of them represents a satisfying solution for Web applications. This section identifies what they do and do not do, by mapping them onto the trust management infrastructure framework discussed above. This is not to say that a system missing a component is useless, but rather to show how to add its missing pieces or to show how it can collaborate with other systems toward in building a general trust management system.

2.3.1 PICS

PICS, which stands for Platform for Internet Content Selection, is both a metadata format and a protocol. The system was originally designed as a technical solution to protect children from pornography on the Internet without suppressing freedom of speech. PICS enables content providers and trusted third parties to rate their sites and parents and supervisors to set filtering criteria for their children based on the ratings.

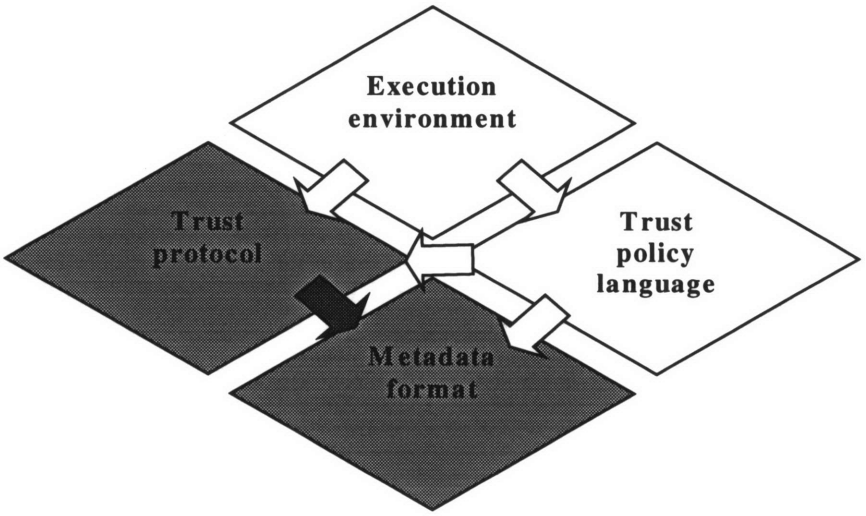

Figure 3 PICS in the Trust Management Infrastructure

Figure 3 shows a graphical representation of PICS in the trust management infrastructure. The main strength of PICS is its ability to express complex semantics within a machine-readable syntax structure. The trust protocol is relatively straightforward; there are ways to distribute labels and methods for querying them with certain attributes. The original PICS specification has neither a policy language nor an execution environment, although there is a proposed policy language called PicsRULZ described in detail in Section 4.2.

It turns out that many other applications have the same need for a general metadata format to make rich assertions as PICS does in content selection. For example, the digital library community needs metadata for cataloging books and journals. The privacy community needs metadata for labeling Web sites in terms of privacy practice. The security community needs metadata to express the semantics of a digital signature. PICS has moved quickly to become the standard, general metadata format on the Internet, with some modifications from its original specification.[1]

The lack of a trust policy language and an execution environment limit the extent PICS can apply to Web applications requiring trust. PICS may be sufficient for applications involving content selection, where application-specific, proprietary policy language and execution environment can select content by matching PICS labels against user policies for acceptable ratings. However, PICS may not be applicable for applications involving database search based on PICS labels. One deficiency in content selection applications is that a fraction of the "hits" from a normal search engine will not comply with a user's policy for content selection and a user must test each "hit" to ensure compliance. A more efficient method is to give the user's policy for content selection to a search engine and the engine would return only hits that comply with that policy. In order to facilitate this database search application, both the clients and the search engines must agree on certain open-standard policy languages for PICS. In addition, both sides also need to have general execution environments to handle possibly various trust policy languages, or various metadata formats beside PICS, or various protocols to negotiate the transfer of the client's policy.

PICS alone does not provide the complete solution for managing trust, and it does not need to do so. Rather, the rich assertion system in PICS is a valuable building block in the trust management infrastructure. Other protocols and policies can simply take PICS as a component and build on top of it, as the PicsRULZ and Profiles-0.92 policy languages have already done.

2.3.2 X.509

X.509 [CCITT88b] is a standard for authenticating users in an X.500 directory server [CCITT88a]. It is often referred to as an *identity certification scheme*, because the certificate is a signed statement that maps an identity to a public key. X.509 has a simple metadata format to express identity and a simple protocol for requesting a set of certificates. Figure 4 shows how X.509 maps to the trust management infrastructure.

[1] There is a Next Generation of PICS (PICS-NG) Working Group in W3C, whose goal is to create a next generation of PICS label format.

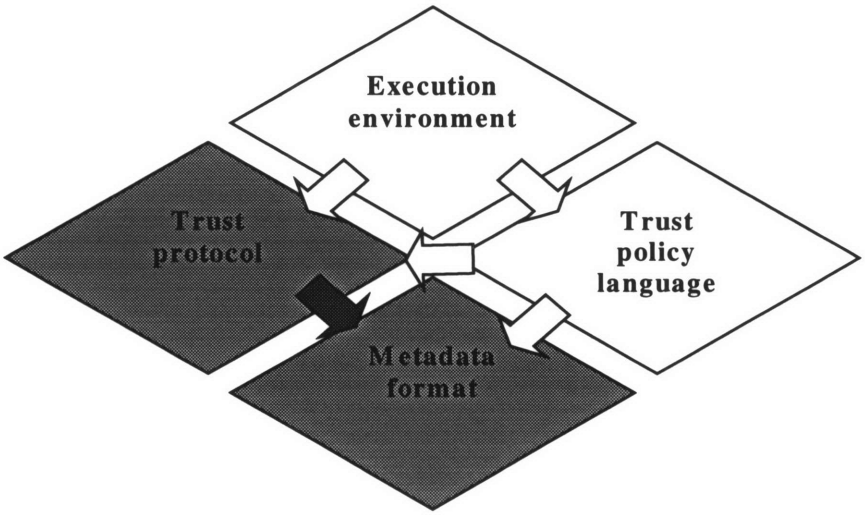

Figure 4 X.509 in the Trust Management Infrastructure

The metadata format in X.509 is called an identity certificate. It contains seven static fields encoded in ASN.1: version number, serial number, signature algorithm and signature bits, certificate issuer, validity period, name of the subject, and the public key information of the subject. The metadata format cannot be extended to carry additional information about who the certificate issuers are or what the certificate is authorized to do.

The certification structure in X.509 is hierarchical. The root is called the Internet Policy Registration Authority (IPRA). Beneath the IPRA are Policy Certification Authorities (PCA), each of which establishes and publishes its policies for registration of users or organizations. PCAs in turn certify CAs, which in turn certify subordinate CAs, users, or organizations.

The trust protocol in X.509 is simple and is based on the hierarchical certification structure. When user A wants to authenticate user B, user A finds the proper certification path by traversing up the certification hierarchy until a mutual CA is reached and then traversing down the hierarchy until user B is reached.

As mentioned in Section 2.2, X.509 does not have a trust policy component. The certification path is simply a data structure for trust delegation in a certification structure. Interpretation of the information conveyed by this algorithm requires a mechanism that is extrinsic to X.509.

X.509 standard is good at authenticating public keys, but that alone is not enough for most Web applications. For example, in a Web application which approves on-line purchase order in a company Intranet, it is not enough to authenticate the purchase order form. The application needs to have a trust policy language to specify who are authorized to place certain purchase orders and an execution environment to evaluate these policies. Authenticating a purchase form is simply not enough.

From the standpoint of the trust management infrastructure, X.509 is similar to PICS. They both contain and omit the same component. Functionally, PICS conveys information about an information resource, and X.509 conveys information about an

entity. Both of them have wide user bases and are the important steps toward a general trust management infrastructure.

2.3.3 PolicyMaker

PolicyMaker [BFL96] was the first system to take a comprehensive approach to trust problems independent of any particular application or service. It has a general metadata format ("credentials"), a trust policy language, and an execution environment. As indicated in Section 2.1, PolicyMaker does not deal with the trust protocol component of what I call the trust management infrastructure. Figure 5 shows a graphical representation of PolicyMaker in the trust management infrastructure.

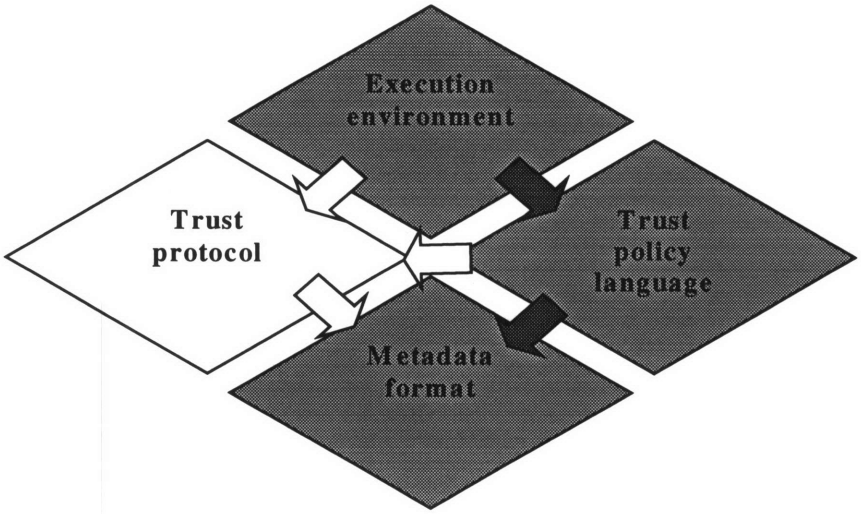

Figure 5 PolicyMaker in the Trust Management Infrastructure

PolicyMaker has its metadata format called *credentials*[2]. It broke new ground by treating credentials as programs. A credential is a type of "assertion." It binds a predicate, called a filter, to a sequence of public keys, called an authority structure. The form of an assertion is:

```
Source ASSERTS AuthorityStruct WHERE Filter
```

Here, *source* indicates the source of authority, generally a public key of an entity in the case of a credential assertion. *AuthorityStruct* specifies the public key or keys to which authority is granted. *Filter* specifies the nature of the authority that is being granted. Both *AuthorityStruct* and *Filter* are represented as programs to maximize their generality. For example, the following PolicyMaker credential

```
pgp:"0x01234567abcdefa0b1c2d3e4f5a6b7"
  ASSERTS
     pgp:"0xf0012203a4b51677d8090aabb3cdd9e2f"
  WHERE
     PREDICATE=regexp:"From Alice";
```

[2] PolicyMaker credential syntax has evolved since [BFL96] was published. Readers should consult [BFRS97] for up to date information.

indicates that the source PGP key "`0x01234567abcdefa0b1c2d3e4f5a6b7`" asserts that Alice's PGP key is "`0xf0012203a4b51677d8090aabb3cdd9e2f`".

Another major innovation in PolicyMaker is the decision to make credentials and policies the same type of object. A policy is also an assertion. The only difference is that policies are unconditionally trusted locally, and credentials are not. The *source* field in a policy assertion is just the keyword "POLICY", rather than the public key of an entity granting authority. Credentials are signed assertions, and the public key in the *source* field can be used to verify the signature.

Both credentials and policies are interpreted within a safe PolicyMaker execution environment. A PolicyMaker engine has no need to make any network connection, because it does not run trust protocols. Therefore, one property of a "safe" execution environment is limited network and resource access. The PolicyMaker engine is not dynamically extensible; if an unknown language is encountered during execution, PolicyMaker does not install software modules dynamically.

Because it does not have a trust protocol, PolicyMaker requires that the host application send all relevant credentials at the time it submits a request. This is considered a drawback, which limits its applicability in the context of the World Wide Web. The drawback can be illustrated in the following trust policy, which finds an authorized PICS label to make assertions about a particular Web object:

```
Retrieve the Web object first.  If there is an embedded PICS label
and the label is rated by entity A, B, or C, return that label (the
label is authorized) and exit.  If the label is rated by a unknown
entity, query an auditor D for an endorsement of that entity.  If
the entity is endorsed by D, return the label and exit.  If no label
is found authorized, retrieve labels from bureau E and F in turn and
process the labels as if they were embedded labels.
```

Clearly, an application evaluating the policy above cannot predict the "right" set of credentials and retrieve them prior to the policy evaluation, an assumption of the PolicyMaker approach. Moreover, the "right" set of credentials changes constantly with respect to the state of the Web object, the label bureaus, the auditor, and the network connection. The only way to evaluate that policy correctly and efficiently is to put the PICS trust protocol under policy control. REFEREE, as a successor of PolicyMaker, realized this deficiency early in the design phase, and it put trust protocols under policy control.

2.3.4 Microsoft Authenticode

Microsoft Corporation was the first, among its competitors, to create a standard to tackle a particular trust management problem called code signing (see Section 2.4.1). Authenticode provides users with the assurance of accountability and authenticity for software downloaded over the Internet. The proposal went public in April 1996 [MS96], and the system was available in Microsoft Internet Explorer 3.0 in October 1996. Authenticode uses PKCS#7 [RSA97] and X.509 as the metadata format, X.509 as the trust protocol, a graphical interface to specify user policies and an implicit, non-extensible execution environment to evaluate user policies and run trust protocols.

Below is a graphical representation of Authenticode in the trust management infrastructure.

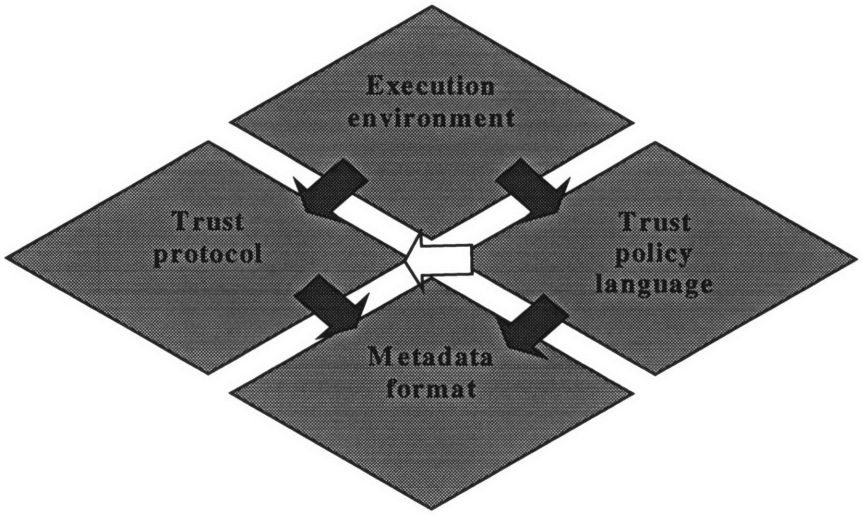

Figure 6 Authenticode in the Trust Management Infrastructure

Authenticode is a complete trust management infrastructure in the sense that all components work together to solve a particular variation of "code-signing" problem. The drawbacks include:

Limited metadata format

Authenticode uses the PKCS#7 signature format and the X.509 identity certificate as the accepted metadata formats. The PKCS#7 signature format is used for making assertions about the software. The X.509 identity certificate is used for authenticating the PKCS#7 signature format. No other metadata formats are allowed.

Limited trust protocol

Authenticode uses the standard X.509 protocol to get a set of certificates for authentication purpose, as described in Section 2.3.2. No other trust protocols are allowed.

This single standard protocol imposes a serious constraint on the generality and applicability of Authenticode. For users who do not trust the X.509 authentication scheme, Authenticode becomes a useless system. For users who want to use Authenticode, they must accept all properties in X.509 as part of their trust policies. And if one day X.509 is compromised (e.g. if a root key is stolen), Authenticode will go down with it.

This analysis shows that it is almost a necessity to allow multiple trust protocols in the trust management infrastructure. It not only gives users the freedom to choose what they trust, but it also eliminates the single point of failure in an interconnected infrastructure of trust.

Limited trust policy

The trust policy in Authenticode is limited by the expressiveness of the graphical interface in the Internet Explorer. Below is a tour of the interface followed by discussion of its advantages and drawbacks.

Authenticode separates individual software publishers from commercial software publishers to distinguish between hobbyist code publishers and professionals. If a user receives a program signed by an unknown entity, Authenticode prompts the user for permission. Figure 7 shows the graphical interface for getting users' permission. On the left, the window interface prompts users to install a Microsoft program from a commercial software publisher. On the right, the window prompts users to install Dan Ziembienski's software from an individual software publisher.

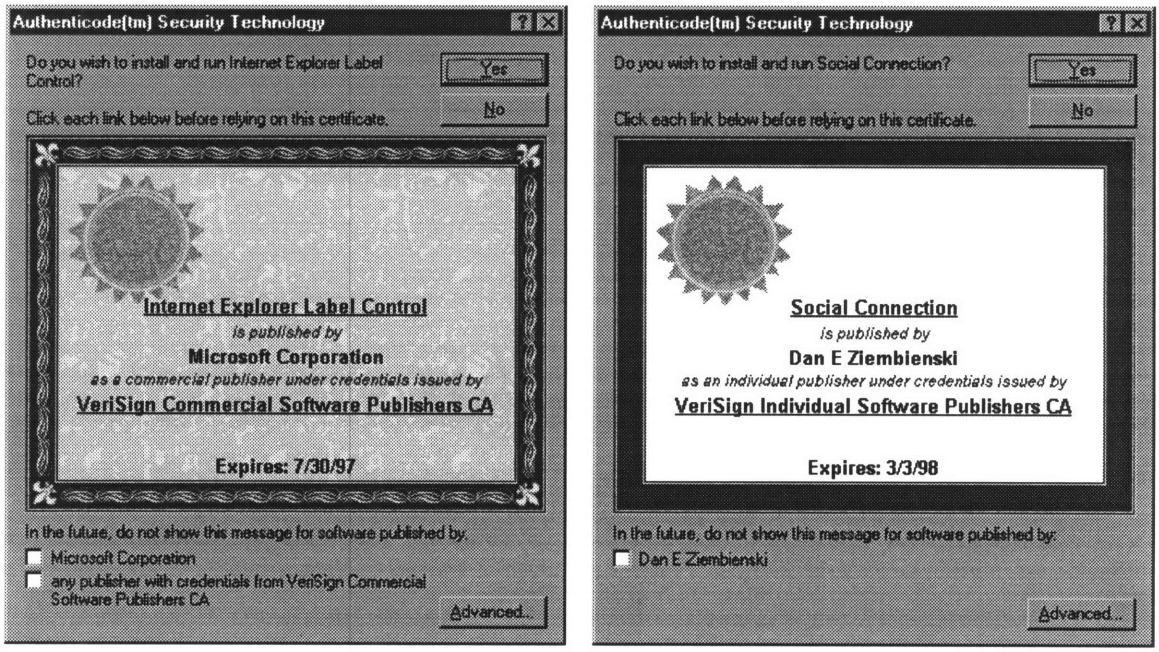

Figure 7 Authenticode User Permission Interface

Inside the windows there are links to more information about the publisher, the software, and the certification authority. In Figure 8, users can configure their trust policies to trust a list of named entities unconditionally (i.e. "don't prompt me"). A checkbox lets users trust all commercial software publishers certified by a named CA.

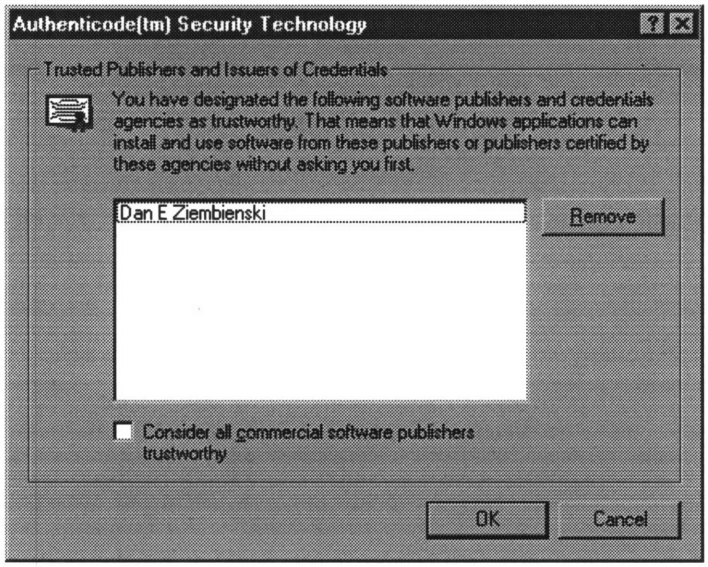

Figure 8 Configuring a List of Trusted Entities in Authenticode

The advantage of this "interface driven" trust policy configuration is the ease of use. The drawback is the lack of expressiveness. For example, such policy schemes cannot do trust delegation. As an example in Figure 7, most users would not know Dan Ziembienski or whether he is trustworthy to publish software on the Internet. A solution is to delegate trust to a trusted auditor, who would vouch for the publishers that are considered trustworthy under the auditor's judgement. As long as the users trust the auditor, they can simply query the auditor once they encounter an unknown publisher. Trust delegation is just one of the many scenarios users will need to express their complex trust relationships. It is my belief that a carefully crafted policy language should come first in the design order and that the graphical interface should build on top of it.

Limited applicability

The inability to do trust delegation in Authenticode blurs the distinction between authentication and authorization from the application point of view, which imposes a serious limitation on its applicability. Given that all authenticated entities look alike, there is no way for an application to assign different authorizations. It is ok for applications where authorizations are trivial, such as electronic mail where anyone is authorized to write his or her own email messages. It is not ok for applications where authorizations are non-trivial, as in code signing where some content providers may not have the authorization to run their software in some users' local environments.

On the spectrum of simplicity versus generality in a system design, Authenticode sits at the simplicity end of the spectrum, and my thesis sits at the generality end of the spectrum. Authenticode technology is good in the sense that it forces both the content providers and the end users to face the problem of trust management in code signing. However, Authenticode does not adequately address the code signing problem or other trust management problems. It is my belief that when application developers have growing needs for trust management for their particular Web applications, they will

realize the limitation on what Authenticode can do and demand a more general framework.

2.4 Examples of Trust Management Problems in the WWW

Trust management problems exist in many Web applications. This section presents different types of trust management problems and identifies their complexities.

2.4.1 Code Distribution

When she was surfing the Web, Alice found the following Web page:

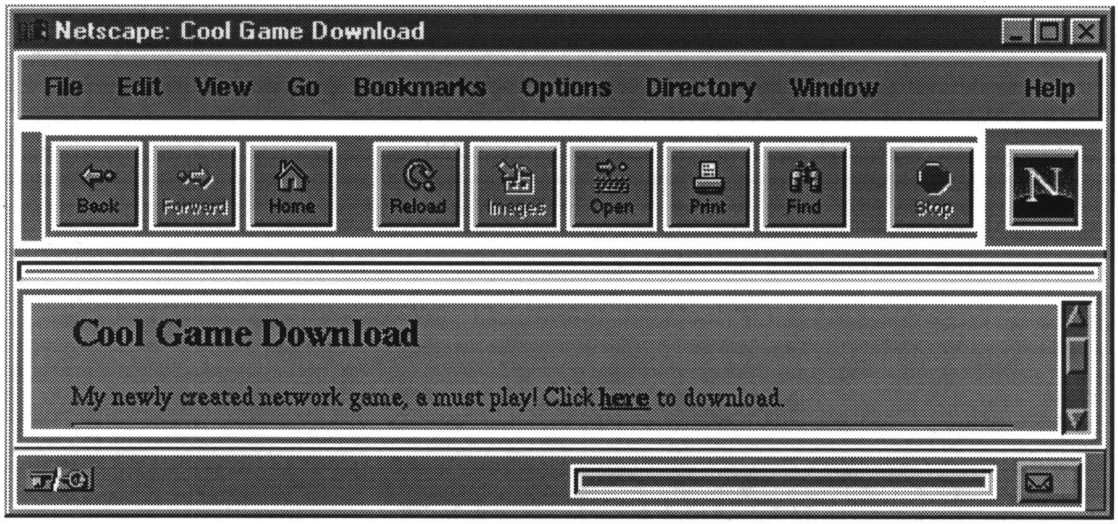

Figure 9 Cool Game Download

Despite how "cool" the game is, Alice has concerns that prevent her from downloading the game on her machine:

Does this game contain a virus that would erase her hard drive? (security issue)
Does this game secretly collect information from her computer? (privacy issue)
Does this game run in her MacOS with 16MB of RAM? (capability issues)
Is this game fun to play? (content issue)
Whom should she trust to make assertions about this cool game? (trust delegation issue)
Does this code come from the author or the trusted sources? (authentication issue)
Has the code been altered during transmission? (integrity issue)

These concerns are all parts of the trust management problem in code distribution. With the growing popularity of Internet access and higher network bandwidth, the active code distribution channel has moved away from the shrink-wrapped model toward network distribution. Typical examples of active code on the Web are Java applets, Netscape plug-ins, Microsoft ActiveX controls, freeware, shareware, and commercial software, software patches, and even macros in static documents[3]. The traditional shrink-wrapped

[3] Macro is considered active code, and active code can be malicious in the form of a virus. An example of a macro virus is "Winword Concept" in Microsoft Word documents [NCSA95]. The virus is automatically

model, which establishes authentication through a branded cover from a known distribution channel (such as a neighborhood computer store) and establishes integrity through a tamper-proof seal outside software diskettes, no longer applies, and an alternative approach is needed to establish trust.

There are currently three approaches to deal with untrusted code. The first is the *sandbox* approach, used in the Java Virtual Machine. The Java sandbox restricts untrusted Java applets to perform a limited set of actions in the host computer. This "trusting no one but yourself" model is ideal in theory but hard to achieve in practice. The sandbox is very hard to make 100% bulletproof, from the engineering perspective. No matter how much effort is put into building a robust sandbox, hackers can always find security holes to compromise the sandbox[4]. Moreover, the restrictions in the sandbox seriously limit what Java applets can do. For example, Java 1.0.x applets can make connection only to their originating host. Programmers cannot create applets that are, for example, groupware or network games, where connections to other sites on the Internet are needed. The new release of the Java 1.1 specification patched this deficiency by granting signed applets more access on top of the sandbox; this utilizes the code-signing approach described below.

The second approach is *proof-carrying code* developed at the Carnegie Mellon University [Necula97]. It is a software mechanism that allows a host application to determine with certainty that code is safe to execute regardless of where it came from. For this to be possible, the code author or the third party distributor must provide a safety proof that attests to certain code's safety properties. The application can then easily and quickly validate the safety proof without using cryptography or consulting any trusted third parties. The main concern about this approach is applicability. Since the construction and the validation of proof depends on the particular language syntax and semantics, this approach is not practical in view of the number of different types of executable currently used on the Web (Java Bytecode, Java Script, Visual Basic Script, ActiveX Control, ···etc).

The third approach is *code-signing*, which uses cryptography to establish authentication and integrity of an untrusted piece of code. Software vendors or trusted third parties provide digitally signed metadata to express trust, and applications can make local trust decisions based on them. Most current systems support this approach, including PICS with DSig extensions [DLLC97], PKCS#7, and Java JAR. The main advantage in this approach over the previous two is that it not only works with any type of active code, but it also works with any type of object that can be properly described using a metadata format, including static documents, identities, or other metadata.

---

executed whenever an infected document is opened within Microsoft Word. Once active, all documents using the File 'Save As' menu item are automatically infected.

[4] For example, on May 16 1997 a team of researchers at the University of Washington found a verifier bug as part of a research effort developing automatic Java verification services. The team found that JDK1.1.1 Bytecode verifier does not check whether a method allocates enough space to hold the input arguments passed in from a caller. If a method is given more arguments than it has room for in the space allocated to its local variables, this could cause a stack overflow. This most likely leads to the Java VM crashing, but it can potentially be used for malicious attacks. See http://java.sun.com/sfaq for more information.

All three are valid approaches for solving the trust management problem in code distribution, and they can be combined to create an even more robust variation. For example, there is a notion of *fine-grain access control*[5] implementation of Java applets, which combines the code-signing approach with the Java sandbox approach. Traditionally, the outcome produced by the code-signing approach alone is generally one-bit: trusted or not trusted to run in its host application. In a sandbox, the outcome can be different level of access control, which can be safely under a *fine-grained access control* Java sandbox during execution.

Trust management precisely does the job of bringing various approaches together to work under one roof. In the trust management infrastructure framework, the three approaches represent various metadata formats and trust protocols. Trust policy languages glue them together. This example of a trust policy illustrates the possibilities:

```
If the code can be formally proved with these named properties,
execute it with full permission. Otherwise, check with my code
validation service agents.  If two of my trusted agents say the code
is safe, execute with full permission except for accessing my
private directory.  Otherwise, prompt me for my approval to be
executed in the highly restricted environment.
```

The need to establish trust in code distribution is one of the major driving forces behind the recognition and use of trust management. Existing applications such as distributed computing, active networks, and agents rely heavily on establishing trust over an untrusted network [FL97].

2.4.2 Document Authentication

Alice knew that it was time the score of the Game 4 of the NBA finals was out. She sat in her dorm room and retrieved the following document from the Boston Globe Web server:

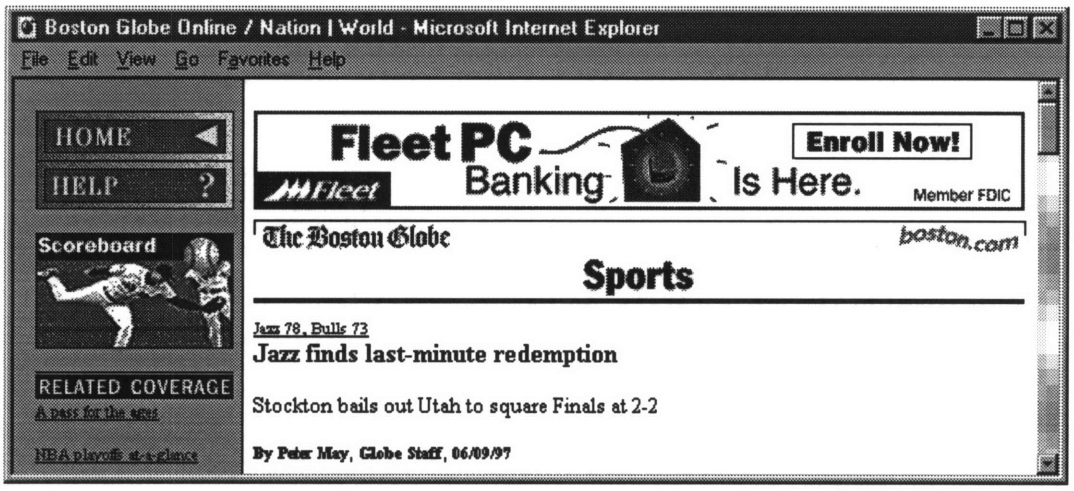

Figure 10 A Snapshot of the Boston Globe Web Document

[5] This concept was advocated by Sun Security Architect Li Gong at the JavaOne conference. See http://java.sun.com/javaone/sessions/slides/TT03/index.html for more information.

The screen showed that that Utah Jazz beat the Chicago Bulls 78 by 73. She was first saddened by the news that her favorite team, the Chicago Bulls, had lost, but then she became suspicious that her neighbor, Bob, a Utah Jazz fan, might have played a trick on her. She needed to know that the article she saw on the screen really came from the Boston Globe (and not from Bob) and that the information was not altered (by Bob) during transmission.

This is an instance of the trust management problem in document authentication. Distributing documents over the network has the benefit of low distribution cost, high bandwidth, and short latency when compared with physical paper in the traditional media. However, the traditional authenticity and integrity properties associated with physical papers are lost when the information is converted into bits and transmitted over the network.

The current mechanism to address the problem of document authentication is digital signatures. A graphical representation of the mechanism is illustrated in Figure 11:

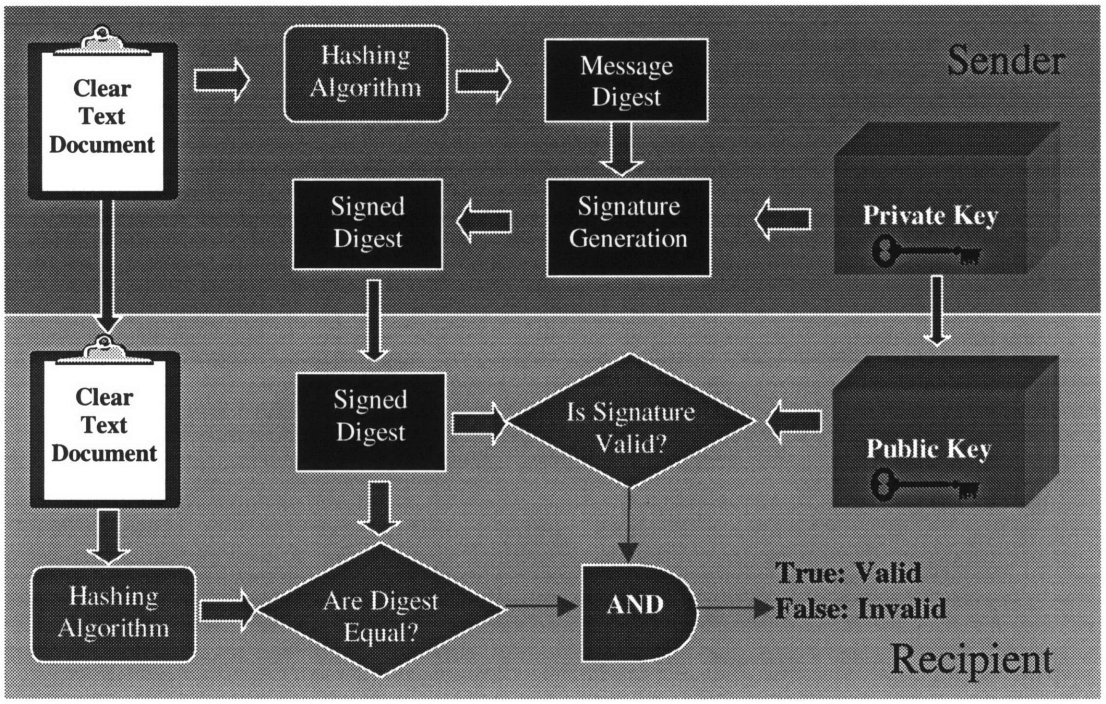

Figure 11 Flow chart for signing and verifying a digital signature

To sign a document, the sender chooses a one-way hash algorithm to compute the message digest of a clear-text document. The digest is then signed using the private key of the sender. Both the clear-text document and the signed digest are transmitted over a potentially untrusted network. The receiver verifies the authenticity and the integrity of the document by checking the signature of the digest and the digest of the clear-text document.

There are many trust problems not addressed by digital signatures alone. First of all, there is a problem in getting the correct public key to verify a sender's digital signature. Public key infrastructures (PKI), such as SDSI, SPKI, and X.509, deals specifically with binding identities to public keys and distributing them securely and efficiently. However,

these PKIs do not offer interoperability among them; problems arise if the sender and the recipient are in different "infrastructures".

Moreover, there are problems in validating digital signatures. Digital signatures come in a variety of attributes, such as the type of algorithms, the number of bits, the source of signatures, the creation dates, and the expiration dates. All the attributes are accountable for determining whether a signature is trustworthy.

In addition, there are problems in determining the semantics of a digital signature. The presence of a digital signature alone has the weakest semantics; namely the entity possessing the key that created this signature has access to the secret key used to generate the signature and the document at the same time [DLLC97]. The semantics of such a signature is usually not strong enough for the recipient to perform useful actions. The signer needs to have a mechanism to express a richer semantics of a signature, such as "I agree with some but not all of this", "I am the second author of the document", or "I verify that all information in the document is true".

Finally, there are trust problems in what a digital signature is authorized to do. For example in a particular bank, a signature from Alice can approve loans up to $10,000, a signature from Bob can approve loans up to $20,000, and when both signatures are present to the bank, they can approve loans up to $50,000.

Trust management precisely addresses these problems mentioned above. The solution to each problem is represented as a single component or a set of interconnected components in the trust management infrastructure. In the trust management infrastructure, a PKI represents an interplay between the metadata format and the trust protocol. The acceptable attributes of a signature represent a local trust policy. The semantics of a signature represent a metadata format. The authorization of a signature represents a trust policy interpreted in a local execution environment.

Document authentication is critical in many Web applications, from Web publishing, to electronic commerce, to national security. In Web publishing applications, a subscriber needs to authenticate the information before reading it. In electronic commerce applications, a consumer needs to authenticate price lists, license contracts, or warranty information before making a transaction with a merchant. In national security applications, a missile needs to authenticate a remote order before it launches to a named target. Digital signatures alone are not enough for these applications and the development of a generic trust management system is necessary for diverse Web applications with the common underlying needs for trust management.

3 Execution Environment

An execution environment is the heart of a trust management system; it is where the local trust policies meet with the rest of the trust management infrastructure, through trust protocols and metadata formats, to make trust decisions as an interconnected entity.

The primary jobs of an execution environment are two: interpret trust policies and administer trust protocols. An execution environment takes requests from its host application, and returns an answer that is compliant with trust policies.

REFEREE is such an execution environment proposed by researchers from AT&T Labs and W3C, including myself. Under REFEREE, trust protocols and trust policies are represented as software modules, which can be invoked and installed dynamically. They can share other's intermediate result through a commonly agreed API. Together they divide up the trust management tasks into pieces, and solve them as a whole. At each level of computation, every aspect of REFEREE is under policy control.

Section one lists the design goals of an execution environment in a trust management system. Section two introduces REFEREE, our proposed solution. Section three and four describes the REFEREE internal architecture and primitive data types. Sections five and six provide a standard procedure to bootstrap and query REFEREE.

3.1 Design Goal

In this section I make a wish list of the properties that a trusted execution environment should have. Some properties may actually be contradictory to each other, and it is up to the system designers to decide which factors are more critical for their targeted applications and intended usage.

General Purpose

The underlying execution environment should be powerful enough to compute all trust decisions users may have. It includes varying degree of complexities of user requests, user policies and third-party credentials. It is conceivable that the underlying evaluation mechanism is Turing-complete to serve its purpose.

Constrainable

Despite how powerful the execution environment can be, the host application should be able to impose constraints on the execution environment. For example, a Web browser may impose certain memory usage, filesystem access, and network access constraints on the environment. The environment needs to propagate and enforce the constraints to its executing policies and protocols.

Extensible

As the trust management infrastructure matures, new trust policies, trust protocols, and metadata formats are introduced where an existing execution environment does not understand. An execution environment should be extensible enough to accommodate new pieces of the components dynamically, instead of returning "policy not understood" or "protocol not understood" answers.

Deterministic

If all the inputs to the execution environment are the same, the same request should return the same answer. The order of evaluation may be different, however.

Platform Independent

The execution environment should be platform independent; it should not rely on specific attributes of the host. This will serve multi-platform operating systems as exists today, such as Windows, UNIX, and MacOS.

Efficient

The environment and its running software modules should be efficient enough so that applications do not see a drastic speed penalty for doing trust management.

3.2 REFEREE

REFEREE is not a standalone application; it must reside in a host application. For the purpose of understanding REFEREE, I model a typical Web application in the following figure and place REFEREE with respect to the model:

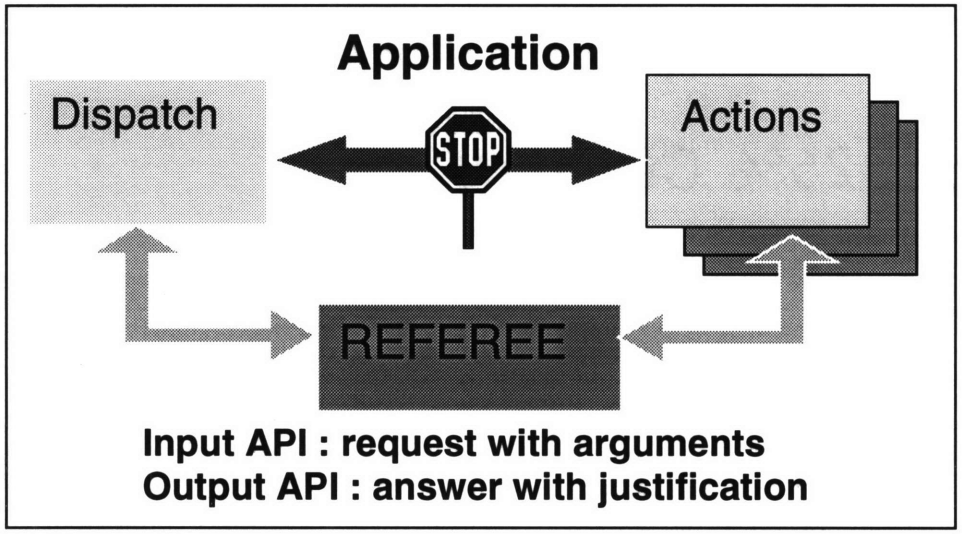

Figure 12 REFEREE External API

The dispatch module is responsible for generating requests. The type of requests depends largely on the context of the applications. For example, in a Web proxy, the request might be "fetch this URL from its source"; in a Web server with access control, the request might be "get this document from the file system"; and in a Web browser, the request might be "execute this Java applet". Formally these requests are dispatched directly to the action modules, where the actions take place. The results of the actions are returned back to the dispatch module.

REFEREE puts a stop sign between the dispatch module and the action module, when potentially dangerous or unauthorized actions are requested. Instead, the dispatch module consults first with REFEREE, through a standard request API. REFEREE invokes the appropriate trust policies and protocols based on the request and its

associated arguments. Then REFEREE returns an answer with some justifications. The dispatch module is then in the position to decide whether to continue the request to the action modules, modify the request and send to REFEREE again, or terminate the request.

From this architecture, it can be seen that REFEREE is *recommendation-based.* The result returned by REFEREE is purely a recommendation to the dispatch module. It is up to the dispatch module to enforce, override, or even ignore REFEREE's recommendation.

3.3 REFEREE Internal Architecture

The REFEREE execution environment is an extensible and self-modifiable execution environment, although it appears to the host application as a monolithic tri-value decision box. The basic computing unit in REFEREE is a *module.* A REFEREE module is an executable block of code that processes input arguments and asserts additional statements. It can also defer subtasks to other modules and make trust decisions based on returned assertions. Together the interconnected REFEREE modules can process requests from the host application and produce a recommendation.

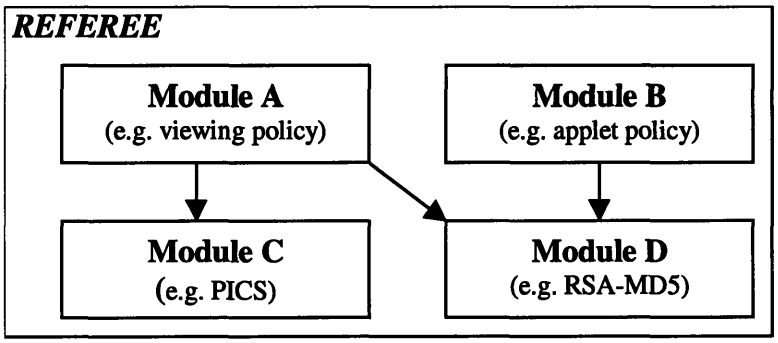

Figure 13 Sample block diagram of REFEREE internal structure.

Figure 13 shows a sample REFEREE execution environment with four modules and their dependency arrows. Module A contains a trust policy for viewing Web pages. Module B contains a trust policy for downloading Java applets. Both A and B call D to verify RSA-MD5 signatures. Module A calls module C to retrieve PICS labels.

The separation of duty among REFEREE modules has several advantages. First of all, existing modules can be updated without affecting other modules, as long as the upgraded modules keep the API backward-compatible. For example, module A and B do not care how module D verifies RSA-MD5 signatures. Therefore module D can be updated with more optimized code without changing module A and B. Moreover, new modules can be introduced dynamically. For example, if module C starts returning PICS labels with DSA-SHA1 signatures that module A cannot verify, module A can upload a module to handle the verification. Other modules in REFEREE, including module B, can then share the new module transparently.

Zooming in, a REFEREE module looks like the following figure:

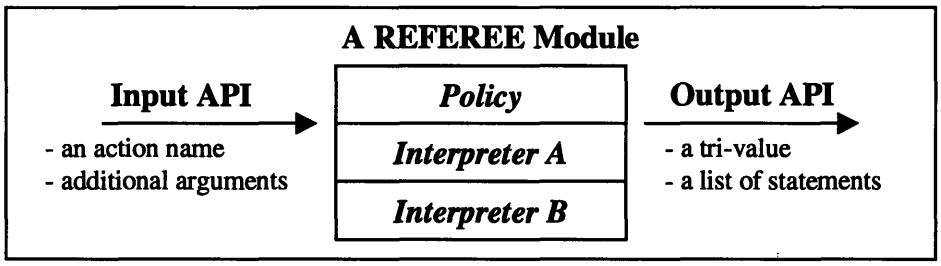

Figure 14 Required interface for every REFEREE module

Each REFEREE module has the same API as REFEREE itself. The input is an action name with additional arguments. The arguments provide additional information about the action, which are unconditionally trusted by the module. If a module deals with content selection, the input arguments may contain the URL of interest, or the public keys of the trusted raters to make assertions about the URL. The output is a tri-value answer with a list of statements as justification. All REFEREE modules must adhere to the same API to ensure interoperability among them.

Internally, a REFEREE module consists of a *policy* and zero or more *interpreters*. The policy is a code fragment written in a high-level policy language, and the interpreters are executable programs for interpreting the policy or other interpreters. The set of interpreters in a REFEREE module is hierarchical; the module policy is interpreted by the highest-level interpreter, which is in turn interpreted by a lower-level interpreter, and so on. The lowest-level interpreter is interpretable by the underlying REFEREE execution environment. In Figure 14, the module policy runs on top of interpreter A, which runs on top of interpreter B, which runs on top of REFEREE.

The separation of policies and interpreters has many advantages. First of all, the same interpreter can run different user policies, as long as they are written in the same policy language. Conversely, the same policy language can be ported to different REFEREE-enabled applications using different interpreters. Moreover, from the security point of view, policies are generally easier to prove correct than interpreters because of the complexity of language constructs. Therefore mutually untrusted parties are more willing to exchange and interpret other's policy preferences using high-level policy languages such as Profiles-0.92 languages (see Chapter 4) than low-level programming languages such as Java. And finally, the REFEREE architecture can accommodate policies written in both high-level languages (for average users) and low-level languages (for sophisticated users) in the same execution environment. I will discuss more on this aspect in Chapter 4.

3.4 REFEREE Primitive Data Types

REFEREE has three primitive data types. They are explained in this section.

3.4.1 Tri-Value

Tri-values are three-valued logic operands. There are three possible values:

- true
- false
- unknown

Notions of *true* and *false* are familiar from Boolean logic. The additional *unknown* value reflects the fact that some trust decisions are neither *true* nor *false*. For example, when asking about authorization of a particular action, such as "should the following purchase order be approved?", there are typically three possible outcomes:

- *true*, meaning "yes, the action may be taken because sufficient credentials exist for the action to be approved."
- *false,* meaning "no, the action may not be taken because sufficient credentials exist to deny the action."
- *unknown*, meaning "REFEREE was unable to find sufficient credentials either to approve or to deny the requested action."

In the third case, the unknown value returned by REFEREE forces the host application to decide what action (if any) should be taken. If the requested action is a purchase order approval, the host application can inform relevant parties for further considerations instead of granting or denying the order.

3.4.2 Statement and Statement-List

Statements are information acquired during the execution of modules. They are the common information interchange container among different modules. All statements are two-element s-expressions. The first element conveys the context of the statement and the second element provides the content of the statement. For example, if a delegation REFEREE module wants to make a statement that "Bob is not trustworthy", it can be expressed in the following statement:

```
( ( "delegation-program" ) ( "Bob" ( trustworthiness 0 ) ) )
```

If the content of a statement expresses an authorization, the context indicates the source of authority. The host application or other REFEREE modules can make more intelligent trust decisions based on not only "what does the statement say" but also "who says it". The use of statements facilitates a dynamic REFEREE execution environment; REFEREE modules can delegate trust evaluations to other modules in REFEREE and know who make the statements, just as applications can delegate trust to third parties on the network and know who make the assertions.

A statement list is an *ordered* list of statements. It generally acts as a transcript of statements that a REFEREE module makes.

3.4.3 Module Databases

A *Module database* binds action names to REFEREE modules, making it possible to call a module by an action name, as is common in almost all programming languages. A module database consists of entries. Each entry is a triplet, *identifier*, *code-fragment*, and *language name*. *Identifier* is a string that uniquely identifies the entry in its local name

space. *Code-fragment* is the actual policy statements or interpreter codes. *Language name* is a string to identify the language the code-fragment is written in and how to interpreter it. An example of a module database is as follows:

identifier	code-fragment	language name
download-applets	<download-policy>	http://www.javasoft.com /jdk1.1/
view-URL	<view-URL-policy>	http://www.w3.org/PICS/ Profiles092/
http://www.w3.org/P ICS/Profiles092/	<Profiles-0.92 interpreters>	http://www.javasoft.com /jdk1.1/

Table 1 A Sample Module Database

To get a policy and interpreter pair from the database, the caller supplies an action name and a list of language names supported by REFEREE. The module database first finds the module policy by matching the action name with the entry identifier. If the match fails, the database returns an error message. If it succeeds, it checks whether the language name is in the list of language names supported by REFEREE itself. If it is, the entry is returned as the policy with no interpreter necessary. If it isn't, the database iteratively searches for lower-level interpreters that can interpret the language, until the lowest-level interpreter is written in a language interpretable by REFEREE itself. Then the policy and a list of interpreters are returned.

For example in Table 1, if the requested action name is "view-URL", then the policy is the code-fragment identified by "view-URL". The associated interpreter is the code-fragment identified by "http://www.w3.org/pub/WWW/PICS/Profiles092", assuming REFEREE supports Java JDK1.1. If the requested action name is "download-applets", then the policy is the code-fragment identified by "download-applets". There is no associated interpreter, since the code-fragment is written in Java JDK1.1.

A module database can selectively install or uninstall bindings to control the availability of the modules, by adding and removing bindings in the database. There is no security mechanism in the module database itself to determine which modules can install, uninstall, or query the database. Rather the module database is passed around as an object, and a caller module can exercise access control by removing certain module database bindings before passing it to a callee module.

The use of module databases in REFEREE facilitates a constrainable execution environment to the granularity of each individual REFEREE module invocation. If a caller does not trust a callee module entirely, a caller module can remove bindings of certain dangerous actions from a callee's module database, such as network access, filesystem access, and user interface access. The callee is therefore unable to perform these dangerous actions in the lifetime of its execution.

3.5 Bootstrapping REFEREE

There are two stages in a lifetime of REFEREE: the bootstrap stage, followed by the query stage. During the bootstrap stage, the host application provides REFEREE enough information to be aware of itself. After the bootstrap stage, REFEREE enters the query stage where the requests are accepted and processed.

There are two pieces of information supplied by the host application during the bootstrap stage:

- trusted assertions
- module database

All bootstrapping information is unconditionally trusted. The trusted assertions are key assertions frequently used by REFEREE operations, such as the root public key, cached credentials and certificates. The module database contains a minimum set of bindings that the host application is expected to use.

3.6 Querying REFEREE

Once REFEREE finishes bootstrapping, it is ready to process queries for its host application. The figure below shows the steps for processing a query with three software modules:

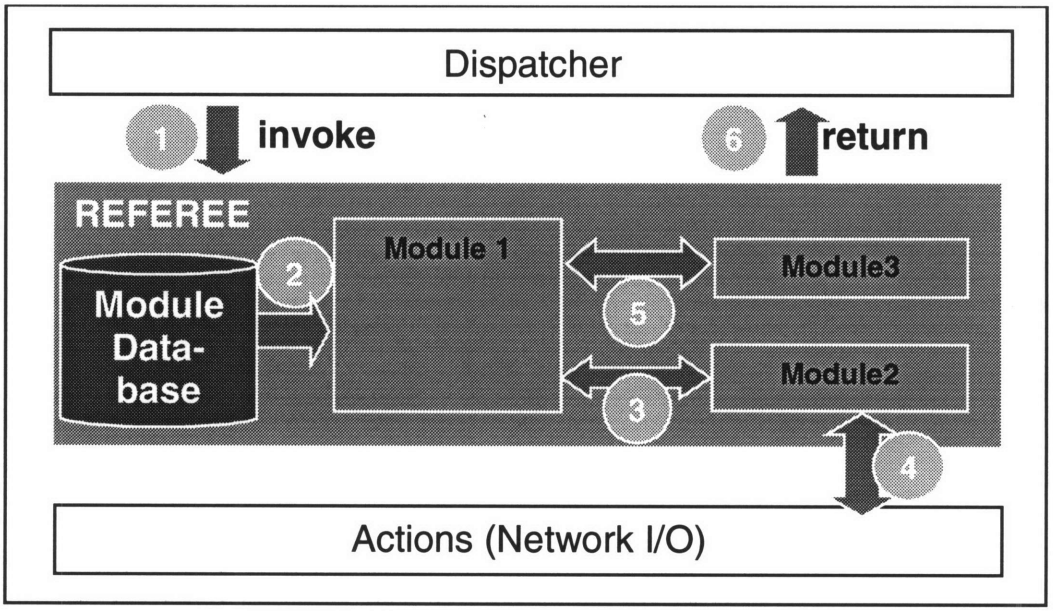

Figure 15 Sample REFEREE Implementation

First, the dispatcher in the host makes a query to REFEREE, which composes an action name with a list of arguments (step 1). When REFEREE receives the query, it grabs the appropriate REFEREE module (module 1) from the module database (step 2), which consists of a policy and an interpreter pair. REFEREE invokes the interpreter with the policy and the list of input arguments. During policy interpretation, the module may invoke other modules (step 3 and step 5), which may in turn call the host-specific actions provided by the host application (step 4). When module 1 finishes the interpretation, REFEREE returns back to the dispatcher with the returned values from module 1.

4 Policy Language

Chapter three shows how the REFEREE execution environment processes queries, interprets trust policies and runs trust protocols in a generic, application-independent way. To prove that REFEREE is indeed a general-purpose execution environment, I implemented two different policy language interpreters as REFEREE modules, namely PicsRULZ and Profiles-0.92.

Both PicsRULZ and Profiles-0.92 describe trust policies based on PICS labels. While PicsRULZ is considerably simpler and easier to use and implement, Profiles-0.92 is more general and expressive. Section one identifies the design goal of a policy language. Sections two and three describe PicsRULZ and Profiles-0.92 in turn. Section four provides four sample policy scenarios and their respective PicsRULZ and Profiles-0.92 translations.

4.1 Design Goals

A policy language describes user policy in a machine-readable format. Despite its simple goal, the design of a good policy language may be more than an engineering task. The more complex the language is, the more expressive the policies can be, at the cost of being more difficult to implement, prove correctness, or build a user interface on top. This section sets asides these engineering tradeoffs, and focuses on the desired properties of a policy language.

Safe

The policy written by a policy language should not cause any undesirable side effect to its host application. That is, assuming the underlying policy interpretation is correct, there should be no way to write a valid policy that crashes the host computer.

Transferable

A profile should be transferable among different applications and platforms. This property allows not only a company to specify a profile for its employees to use on different applications and platforms, but also a user to carry his or her profile to other locations without reconfiguration.

Simple

A policy language should not be a general-purpose programming language (in the sense of Turing-complete), but a simple policy language designed specifically to describe trust policies. However, the language should have an extension mechanism to leave room for future expansion. This property comes hand in hand with the safety and the transferability of a policy language; a simpler language is easier to prove safety and more likely to be executed by an untrusted party when a policy is being transferred.

Well-defined

A policy written in a policy language should be unambiguous irrespective of its specific implementation. It is as if writing a book of law, where the citizens know exactly what is legal and not legal.

Expressive

The language construct should be expressive enough to accommodate realistic policies different users want to specify under different circumstances. The level of expressiveness may depend on programming ability of the people creating the policy, or the complexity of the user interface.

4.2 PicsRULZ

PicsRULZ, by its name, is a rule-based policy language. There are language constructs to write filtering rules based on a requested URL or its associated PICS label attributes retrieved from the author or trusted third parties. It is a simple and concise policy language designed to work with the PICS protocol and metadata format.

PicsRULZ is the result of the PICS Profiles Language Working Group in the World Wide Web Consortium. The language specification is not finalized when the thesis is written. The description of PicsRULZ is based on the draft version presented in a PICS Interests Group Meeting on April 10 in Santa Clara, CA [PICS97c].

PicsRULZ language is organized into clauses. There are seven types of clauses. Some clauses may appear multiple times in a PicsRULZ rule. Although PicsRULZ is a rule-based language, partial evaluation order is enforced to prevent ambiguity. *FailURL* clause takes the highest precedence, followed by *passURL* clause. *Filter* clause is always evaluated last, and other clauses are evaluated arbitrarily between *passURL* and *filter* clauses. By default, if a clause does not specify a returned value, it is assumed that the evaluation continues. All symbols in PicsRULZ are case insensitive, and all quoted strings are case sensitive.

Each of the seven clauses in PicsRULZ is explained below. A complete BNF syntax is in Appendix A.

failURL

FailURL clause is a method to express a list of URL prefixes which are explicitly blocked. If the requested URL matches one of *failURL* list, the evaluation terminates immediately and outputs a *block* answer. For example, the clause

```
failURL ("http://www.harvard.edu" "http://www.caltech.edu")
```

blocks all URLs from Harvard and Caltech Web servers.

FailURL clause takes the highest precedence in the evaluation order. It may appear more than once in a PicsRULZ rule, but it is recommended that they be combined into a single *failURL* list.

passURL

PassURL has the same syntax as *failURL* but opposite in semantics. It means if the requested URL matches one of *passURL* list, the evaluation terminates and outputs

a *pass* answer. *PassURL* takes the second precedence in the evaluation, after *failURL*. As with *failURL*, *passURL* may appear more than once but a single list is preferred. The following clause

```
passURL ("http://www.wellesley.edu")
```

explicitly allows all URLs with "http://www.wellesley.edu" prefix.

serviceinfo

Serviceinfo specifies information about a rating service. There are five attributes in a *serviceinfo* clause: *name*, *shortname*, *bureauURL*, *ratfile* and *bureauUnavailable*. *Name* attribute is the URL of the rating service. *Shortname* binds a local variable to a rating service. *BureauURL* specifies the location of a label bureau to retrieve PICS labels from. *Ratfile* contains either the URL or the actual text of the machine-readable rating description file. *BureauUnavailable* is an exception mechanism to specify what to do when the named label bureau cannot be contacted. Since users may want to utilize more than one rating service for a given URL, multiple *serviceinfo* clauses are allowed in PicsRULZ. All clauses must be evaluated before the *filter* clause because the *filter* clause may use the local name defined in the clause. In the following example,

```
serviceinfo ( name "http://6001.mit.edu/ratings/midterm.html"
              shortname "6001"
              bureauURL "http://6001.mit.edu"
              bureauUnavailable true )
```

the clause retrieves PICS labels rated by "midterm" rating system from "6.001.mit.edu" label bureau. If the label bureau is unavailable, the evaluation terminates and returns *true* (allow).

filter

Filter clause operates on the PICS labels acquired from *serviceinfo* clauses. The clause is divided into two sub-expressions, *pass-expression* and *block-expression*, linked by an explicit *and* logic. That is, a URL passes the filter only if the pass-expression is *true* and the block-expression is *false*. The default value of the *pass-expression* is *true*, and the default value of the *block-expression* is *false*. *Pass-expression* and *block-expression* are composed of *simple-expressions*. A *simple-expression* compares an attribute of a PICS label with a constant (`6.001.grade >= 5.0`), returning a Boolean value (`true/false`). Simple-expressions can be combined with *and* and *or* operators to create arbitrary-depth *pass-expressions* and *block-expressions*. "Not" operator is explicitly omitted for clarity reasons. In the following example,

```
filter ( Pass "(6001.grade = 5.0 or 6001.grade < 2.0)"
         Block "(6001.by = jmiller@w3.org)" )
```

the clause returns *true* (allow) only if the midterm grade is A (to brag about it), or is below D- (to drop the class), and the label does not come from the instructor Jim Miller (he likes to "spam" poor freshmen with fake grades, if students are not confused enough by his lectures).

extension

Extension provides a way to extend the functionality of PicsRULZ. As in PICS-1.1 extension, there are *required* (mandatory) and *optional* extensions. If the extension is required, the rule evaluator must understand and evaluate the extension. Optional extensions need not be evaluated; they are intended for documentation purpose.

In the following example,

```
reqextension ("http://www.w3.org/DSig/RSA-MD5.html")
filter (Block "(6001.by = jmiller@w3.org)"
        Check-signature true)
```

the content of the "RSA-MD5" extension presumably defines a new attribute-value pair `(check-signature <Boolean>)` within a filter clause. Now filter clause returns *true* only if the PICS label contains a valid signature (a `true` value in the attribute) and the signature is not of Jim Miller's. In this case, the use of this extension prevents Jim to fake as another person to bypass the filtering rules.

name

Name clause provides local information about the rule, intended to facilitate the construction of a user interface. There are two attributes, namely *rulename* and *description*. *Rulename* attribute binds the rule to a local name. *Description* attribute is a more detailed description of the rule. For example,

```
name (rulename "6.001-Rule")
      description "Viewing rule for the graded 6.001 midterm")
```

source

Source provides information about where the rule comes from. There are four fields in the clause. *SourceURL* field uniquely identifies the rule. Interested parties can also go to this location to find more information about the rule. *CreationTool* field identifies how the rule is constructed. *Author* field gives an identity (generally an e-mail address) of who creates the rule. *LastModified* gives the date and time that the rule was last modified. An example looks like the following:

```
source (sourceURL "http://web.mit.edu/bendiddle/6001.html"
        creationTool "PicsRULZ-EDITOR/1.0"
        author "bendiddle@mit.edu"
        lastModified "1997.05.06:12.34-0500")
```

Examples of PicsRULZ are in Section 4.4.

4.3 Profiles-0.92

The Profiles-0.92 language [BCKLMR$] was developed in conjunction with the REFEREE trust management system. Profiles-0.92 is a flexible and modular policy language aiming to exploit and demonstrate important features in the REFEREE trust management system.

An instance of a Profiles-0.92 language is a *policy*. A policy consists of an ordered sequence of *rules*. Each rule is represented as an s-expression, in which the first token is an operator, and the rest of the tokens are operands. The evaluation of a policy is top-down. The returned value of the last rule is the returned value of the policy itself. To be easily ported to REFEREE, the returned value of a rule is the same as REFEREE itself, and is the same as REFEREE modules: a tri-value answer with a statement list as justifications.

Profiles-0.92 is rich in expressiveness compared with PicsRULZ. This section highlights six important rule syntax and semantics. Readers should refer to [BCKLMRS] for more detail. A snapshot of the complete language syntax in the modified BNF form is provided in Appendix B.

URL Prefix Matching

```
(url-match URL (<URL-prefix>+) [<prefix-match?>])
```

This function provides a means of explicitly returning a tri-value based on substring matches against particular URLs. The first argument is the symbol `URL`. The second argument is a list of strings to be matched against the given URL. The third argument is a Boolean value which determines whether the string matching should be exact or prefix. If this argument is *true* then URL must exactly match one of the strings for the resulting value to be true. If *false*, one of the strings must be a prefix of URL in order for the resulting value to be true. The `<prefix-match?>` argument is optional; if it is not present it is assumed to be *false* and prefix matching is performed.

For example, the function

```
(url-match URL ("http://web.mit.edu"
                "http://www.wellesley.edu"))
```

returns *true* if the requested URL has any of the listed URLs as a prefix, and otherwise it returns *false*. It is not possible for `url-match` to return an *unknown* tri-value.

The statement-list returned by *URL prefix match* consists of statements of the form `(url-match <URL-prefix>+)` for each relevant URL prefix. In the above example, if "http://web.mit.edu/benbiddle" was requested then the function returns

```
((url-match "http://web.mit.edu"))
```

as the content of the returned statement.

Pattern Matching

```
(match <pattern> <statement-list>)
```

The pattern matching function matches the s-expression `<pattern>` against statements in `<statement-list>`. A match happens when a pattern and a statement are syntactically and structurally equivalent.

In the simplest form, a parenthesis in the pattern matches a parenthesis and a literal element matches a literal element. In addition, there are four special pattern-matching elements:

.	zero or one literal or parenthesized s-expression
*	zero or more literals and parenthesized s-expressions
+	matches one or more literals and parenthesized s-expressions
(RESTRICT operator literal value)	matches some s-expressions of the form (literal value)

Thus, `(* 3 *)` matches `(3)` and `(2 3 4)`, but not `(2 4 5)`. Similarly, `(. (sha-1 +) *)` matches `((foo)(sha-1 3))`, but not `((foo) bar (sha-1 3))`. Quoted strings are matched on a case-sensitive basis; all other elements are matched insensitive to case.

`RESTRICT` pattern-matching elements allow arithmetic comparison on numbers in an s-expression. This is important in the PICS environment, in which a policy may want to test whether the value associated with a transmit-name is less than some threshold value. Arithmetic comparison `operator` can be one of <, >, =, <=, >=, <>, where <> represents "not equal". `Literal` is a symbol (transmit-name in PICS) that identifies the value. A comparison operation happens only if both the pattern and the matching statement have the same literal field. For example, `(RESTRICT < n 3)` matches `(n 2)`, and `(* (RESTRICT < n 3) *)` matches `(foo bar baz (n 2) quux)`, but does not match `(foo bar baz (n 3) quux)`.

If no statements syntactically match the pattern, the returned tri-value is *unknown*. If some statements match and no restrictions are included, the returned tri-value is *true*. If statements match and there are restrictions, the returned tri-value is *true* or *false* depending on predicates in the restrictions. Each comparison operator exists in both normal and "`<operator>!`" form. The presence of an "!" does not modify the matching operation but does change the way the overall match construct computes the returned tri-value. For operators ending in "!", match returns *true* only if every statement that syntactically matches the restriction satisfies the predicate. For non-"!" operators, match returns "true" if any syntactically-matching statement satisfies the predicate. If more than one restriction is present, their *tri-values* are implicitly *and*ed together. If any restriction is false the match returns false. For example, if the statement-list `((n 4) (n 2))` and the pattern is `(RESTRICT < n 3)`, the match would return true, because the second statement `(n 2)` matches the pattern. But if the pattern changes to `(RESTRICT <! n 3)`, the match would return false because not all statements in the matched statement list match the pattern.

The backslash \ character has special meaning within patterns; it is used to quote pattern elements that would normally have special semantics. That is, to match the character + as opposed to one or more s-expressions, use `\+` is used in the pattern. Similarly, the reserved word `RESTRICT` can be escaped with `\RESTRICT` to match the actual symbol instead of the special restrict pattern matcher.

Combinations

```
(and <rule>+)

(or <rule>+)
```

```
(threshold-and <num> <rule>+)
(not <rule>)
(true-if-unknown <rule>)
(false-if-unknown <rule>)
```

Profiles-0.92 provides six tri-value operators. The operators *and*, *or* and *threshold-and* are multi-argument operators and *not*, *true-if-unknown* and *false-if-unknown* are unary operators. Each multi-argument operator takes zero (one for *threshold-and*) or more rules as input. The output tri-value is computed based on the input tri-values, and the output of the statement-list is a concatenation of the input statement-lists. Unary operators work the same way, except that the output of the statement-list is inherited directly from the input. The truth tables for the six operators are provided below.

The *and* operator

The *and* operator is the three-valued version of the Boolean *and* operator. Table 2 describes the operation of *and* when it is given two arguments. The first row represent the truth value for the first argument, the first column represent the truth value for the second argument, and the rest of the cells represent the result of an *and* operation.

rule1 \ rule2	true	unknown	false
true	true	unknown	false
unknown	Unknown	unknown	false
false	False	false	false

Table 2 Truth table for the *and* operator

The *and* operator can take any number of arguments. For more than two arguments, *and* operator recursively reduces itself one argument at a time:

```
(and arg1 arg2 ··· argn) = (and (··· (and arg1 arg2) ··· argn)
```

The *and* of a single argument is that argument itself. The *and* of no argument is *true* by definition. If one of the arguments return *false*, the *and* rule terminates and the rule returns a *false*, because further evaluations will not change the outcome of a *false*.

The *or* operator

The *or* operator is the three-valued version of Boolean *or* operator. Table 3 describes the operation of *or* when it is given two arguments:

rule1 \ rule2	true	unknown	False
true	true	true	True
unknown	true	unknown	Unknown
false	true	unknown	False

Table 3 Truth Table for the *or* operator

As *and* operator, *or* operator can take any number of arguments, and they are recursively reduced if more than two arguments are present. The *or* of a single argument is that argument itself. The *or* of no arguments is false by definition. If

one of the arguments is evaluated to be true, the *or* terminates and returns a *true*, because further evaluation does not change the outcome of a *true*.

The *not* operator

The *not* operator is the three-valued version of Boolean *not* operator. It takes exactly one argument. Table 4 describes the operation of the *not* operator:

	output
true	false
unknown	Unknown
false	true

Table 4 Truth Table for the *not* operator

The *true-if-unknown* operator

The *true-if-unknown* operator is a projection function from three-valued logic to Boolean logic. It takes exactly one argument:

	output
true	true
unknown	true
false	false

Table 5 Truth Table for the true-if-unknown operator

The *false-if-unknown* operator

The *false-if-unknown* operator is also a projection function from three-valued logic to Boolean logic. It takes exactly one argument:

	Output
true	true
unknown	false
false	false

Table 6 Truth Table for the false-if-unknown operator

The *threshold-and* operator

The *threshold-and* operator implements "any m of n" semantics on a list of three-valued arguments. The *threshold-and* operator takes at least one argument, the threshold value as a non-negative integer. A call to *threshold-and* looks as follows:

```
(threshold-and threshold arg1 arg2 arg3 ··· argn)
```

Let n_T, n_F and n_U be, respectively, the number of arguments to *threshold-and* `arg1···argn` that evaluate to *true*, *false*, and *unknown*. We have $0 <= n_T, n_F, n_U, <= n$, and further $n_T + n_F + n_U = n$. Then the value of *threshold-and* is computed as follows:

- if $n_T >= $ threshold, return *true*.
- else if $n_T <$ threshold and $n_T + n_U >=$ threshold, return *unknown*.
- else if $n_T + n_U <$ threshold, return *false*.
- by definition, `(threshold-and 0)` evaluates to *true*.

Invocations

```
(invoke <policy-name> <statement-list> <additional-args>*)
```

Invoke calls the policy named `<policy-name>` with a copy of the `<statement-list>` and possibly some other additional arguments. When the called policy returns, its returned value is a pair consisting of a tri-value and a statement-list. By convention, the caller module appends the returned statement-list of the callee to its internal statement-list. For every statement in the returned statement-list, Profiles-0.92 prepends the name of the called policy to the context of the statement, and appends the statement to the original statement-list that was referenced in the call to invoke. The returned value of the `(invoke ...)` construct is a pair of the tri-value returned by the called policy and the tagged statements appended to the statement-list.

Installations

```
(install-policy <statement-list>)
(install-interpreter <statement-list>)
```

Recall that in Profiles-0.92, there are two types of entries in a REFEREE module database: policy and interpreter. *Install-policy* creates policy bindings in the module database and *install-interpreter* creates interpreter bindings in the module database. In both cases, the information required to make these bindings are passed within a statement-list containing a single statement, and they are of the form:

```
((<context>) (<identifier> <code-fragment> <language-name>))
```

Local Variable Binding

```
(let (<binding>+) <rule>+)
```

Let creates a new sub-environment of the current execution environment and creates in the sub-environment new variable-value bindings. The created local bindings are listed in the list of bindings `<binding>+`. Each `<binding>` is a list of the form: `(<var> <expression>)`. The variable `<var>` is bound to the result of evaluating `<expression>`. Each `<expression>` may optionally be null, in which case the variable is defined but its value is unassigned. The bindings remain in effect for the scope of the *let* rule.

A Profiles-0.92 policy is invoked with a list of argument. Each argument is bound to a local name at the beginning of the evaluation. The first two arguments are mandatory in Profiles-0.92, and are bound to the local name STATEMENT-LIST and URL, respectively. Optional statements are bound to the local names ARG3, ARG4, and so on.

4.4 Sample Policies

This section presents four sample policies with varying complexities. The policies are first explained in English, then translated to PicsRULZ and Profiles-0.92 languages. The section also presents certain variations of the policies expressible only by Profiles-0.92.

The following examples use code-signing as the target trust management problem. The request is "should I download the active content at this URL". The policy returns *true* (or

allow) meaning "yes, go ahead and download", *false* (or fail) meaning "no, don't download", or *unknown* (in Profiles-0.92 only) meaning "prompt me for my attention". Most examples use a hypothetical PICS rating service called *CodeSafety*, with two dimensions, "stability" and "virus", and values from 0 to 10 along each dimension. Higher value designates more stable and less possibility of virus in the described active code. Other examples use "Endorse" and "Multimedia", whose dimensions and values are explained as needed.

4.4.1 Sample policy 1: determine Access Based on the URL

This policy determines access based on requested URL. Such policy is useful if the application knows *a priori* a list of sites or directories it should not download codes from.

Policy in English

```
Do not download the code if the URL is served from Harvard or CalTech
Web servers.
```

PicsRULZ

```
(PicsRule-1.0
  (failURL ("http://www.harvard.edu" "http://www.caltech.edu")
   filter (pass "Unless-Prohibited")))
```

Profiles-0.92

```
(not (url-match URL ("http://www.harvard.edu"
                     "http://www.caltech.edu")))
```

This kind of policy can be very effective in practice. Firewall vendors can compile a blacklist of Web sites that serve potentially dangerous active codes, and place the list in clients' firewalls.

Profiles-0.92 is capable of taking the policy a step further by returning *unknown* if the requested URL is neither in the blacklist nor in the whitelist (sites know to be trustworthy):

Policy in English

```
Do not download the code if the URL is served from Harvard or CalTech
Web servers.  Download it automatically if served from MIT.  Prompt me
for my attention otherwise.
```

Profiles-0.92

```
(theshold-and
   2
   (not (url-match URL ("http://www.harvard.edu"
                        "http://www.caltech.edu")))
   (url-match URL ("http://web.mit.edu"))
   unknown)
```

The blacklist and whitelist ensure good automation of the trust decision process if the lists are reasonably complete. User intervention is needed only when the given URL is in neither the blacklist nor the whitelist.

4.4.2 Sample policy 2: determine access based on PICS labels

This policy requests PICS labels from trusted sources and processes the labels based on the attributes of the received labels.

Policy in English

```
Get PICS labels from MIT and CMU label bureaus.  Download the code if
any of the received PICS labels says the virus is checked with good
confidence.
```

PicsRULZ

```
(PicsRule-1.0
  (serviceinfo (name "http://web.mit.edu/ratings/CodeSafety.html"
                shortname "Safety"
                bureauURL "(http://bureau.mit.edu
                           http://bureau.cmu.edu)"
   filter (Pass "(Safety.virus > 8)")))
```

Profiles-0.92

```
(invoke "load-label" STATEMENT-LIST URL
        "http://web.mit.edu/ratings/CodeSafety.html"
        ("http://bureau.mit.edu" "http://bureau.cmu.edu"))
(match (("load-label")
        (((version "PICS-1.1") *
          (service "http://web.mit.edu/ratings/CodeSafety.html") *
          (ratings (RESTRICT > virus 8)))))
       STATEMENT-LIST)
```

These two policy descriptions have the same semantics, but Profiles-0.92 looks a bit more complex, especially in its pattern-matching language. This is again the classical trade-off between expressiveness versus elegance in language design.

The expressiveness in Profiles-0.92 comes in handy when the policy requires **both** labels from both sources to make acceptable assertions about virus. A Profiles-0.92 policy simply changes the match operator from ">" to ">!".

Profiles-0.92

```
(invoke "load-label" STATEMENT-LIST URL
        "http://web.mit.edu/ratings/CodeSafety.html"
        ("http://bureau.mit.edu" "http://bureau.cmu.edu"))
(match (("load-label")
        (((version "PICS-1.1") *
          (service "http://web.mit.edu/ratings/CodeSafety.html") *
          (ratings (RESTRICT >! virus 8)))))
       STATEMENT-LIST)
```

Another useful policy is to have equally trustworthy but potentially conflicting assertions to "vote" among themselves. The policy below creates a "majority wins" among three PICS labels from three different sources, by using *threshold-and* operator in Profiles-0.92 pattern matching language.

Profiles-0.92

```
(let (LabelA (invoke "load-label" STATEMENT-LIST URL
                     "http://web.mit.edu/ratings/CodeSafety.html"
                     ("http://bureau.mit.edu"))
      LabelB (invoke "load-label" STATEMENT-LIST URL
                     "http://web.mit.edu/ratings/CodeSafety.html"
                     ("http://bureau.cmu.edu"))
      LabelC (invoke "load-label" STATEMENT-LIST URL
                     "http://web.mit.edu/ratings/CodeSafety.html"
                     ("http://bureau.wellesley.edu"))
     (threshold-and 2
       (match (("load-label")
               (((version "PICS-1.1") *
                 (service "http://web.mit.edu/ratings/CodeSafety.html")
                 (ratings (RESTRICT > virus 8)))))
               LabelA))
       (match (("load-label")
               (((version "PICS-1.1") *
                 (service "http://web.mit.edu/ratings/CodeSafety.html")
                 (ratings (RESTRICT > virus 8)))))
               LabelB))
       (match (("load-label")
               (((version "PICS-1.1") *
                 (service "http://web.mit.edu/ratings/CodeSafety.html")
                 (ratings (RESTRICT > virus 8)))))
               LabelC))))
```

4.4.3 Sample Policy 3: Determine Access Based on Multiple PICS Labels and Sources

This sample policy combines the previous two sample policies to create a more complex, but more realistic policy. It specifies which URL prefixes are unconditionally allowed and blocked. For the unspecified URL prefixes, the policy determines access based on PICS labels from various sources with various ratings schemas.

Policy in English

```
Do not download the code if the requested URL comes from Harvard Web
site or from Bendiddle's directory at MIT.  Any other content served at
MIT Web site can be downloaded.  For all other Web sites, you must get
PICS labels from MIT label bureau with MIT safety rating and from
Wellesley label bureau with PCWorld Multimedia rating.  Download the
code if the labels assert good virus check (v > 8) and have cool sound
and video (s >= 5, v >=5).
```

PicsRULZ

```
(PicsRule-1.0
  (failURL ("http://www.harvard.edu" "http://web.mit.edu/bendiddle")
   passURL ("http://web.mit.edu")
   name (rulename "Download-Code"
         description "This rule is created for thesis illustration …")
   source (sourceURL "http://www.w3.org/PICS/TrustMgt/Rules/Code.html")
   serviceinfo (name "http://web.mit.edu/ratings/CodeSafety.html"
                shortname "Safety"
                bureauURL "http://web.mit.edu")
   serviceinfo (name "http://pcweek.com/ratings/Multimedia.html"
                shortname "Multimedia"
                bureauURL "http://www.wellesley.edu")
   filter (pass "(Safety.virus > 8)"
           block "((Multimedia.sound < 5) or
                   (Multimedia.video < 5))")))
```

Profiles-0.92

```
(and (not (url-match URL ("http://www.harvard.edu"
                          "http://web.mit.edu/bendiddle")))
     (or (url-match URL ("http://web.mit.edu"))
         (let (SafetyLabel
                 (invoke "load-label" STATEMENT-LIST
                         "http://web.mit.edu/ratings/CodeSafety.html"
                         URL ("http:// http://web.mit.edu")))
              (MultimediaLabel
                 (invoke "load-label" STATEMENT-LIST
                         "http://pcweek.com/ratings/Multimedia.html"
                         URL ("http://www.wellesley.edu"))))
         (and
           (match
             (("load-label")
              (((version "PICS-1.1") *
                (service "http://web.mit.edu/ratings/CodeSafety.html")
                * (ratings (RESTRICT > virus 8))))) SafetyLabel)
           (match
             (("load-label")
              (((version "PICS-1.1") *
                (service "http://pcweek.com/ratings/Multimedia.html")
                * (ratings (RESTRICT >= video 5))))) MultimediaLabel)
           (match
             (("load-label")
              (((version "PICS-1.1") *
                (service "http://pcweek.com/ratings/Multimedia.html")
                * (ratings (RESTRICT >= sound 5))))) MultimediaLabel)))
```

PicsRULZ has a more elegant policy over Profiles-0.92 due to its implicit *and* and *or* operations in some clauses. More specifically, *failURL* clause has an implicit *and* with the rest of the clauses in the rule, *passURL* has an implicit *or* with the rest of the clauses except *failURL*, and *filter* has an implicit *and* for the *subexpressions* in both *pass* and *block* expressions. These implicit operators become visible when written in Profiles-0.92 language.

4.4.4 Sample Policy 4: Defer Trust Using Extension Mechanism

Sample policy 4 adds a level of sophistication by setting "*who* is trusted" to make an assertion about the active code. The "who", in PICS term, is the author of the label and the rater of the code. In real life, an application may not know all the raters. A more likely situation is that an application would trust a small number of auditors and accept only labels from raters endorsed by the auditors. In this example, the policy authorizes MIT to endorse raters who show good judgements in rating active contents, and both the rater's label and the endorsement label must be properly signed.

Policy in English

```
Download the code from this URL only if a rater says it is virus free
(virus = 10), and that rater is endorsed by MIT as being an above-
average active code reviewer (CodeJudgement > 8). Both the endorsement
and the code safety label must be digitally signed.
```

PicsRULZ

```
(PicsRule-1.0
  (reqextension ("http://www.w3.org/Dsig/PicsEndorsement.html")
   reqextension ("http://www.w3.org/Dsig/PicsSignature.html")
   serviceinfo (name "http://web.mit.edu/ratings/CodeSafety.html"
                shortname "Safety")
   serviceinfo (name "http://web.mit.edu/ratings/endorsement.html"
                shortname "Endorse"
                bureauURL "http://web.mit.edu"
                endorses "Safety")
   filter (check-signature "(Applet Endorse)"
           Pass "((Safety.virus = 10) and
                  (Endorse.by "mailto:endorsement@mit.edu")
                  (Endorse.CodeJudgement > 8))")
```

Profiles-0.92

```
(and
  (let
    (SafetyLabel (invoke "load-label" STATEMENT-LIST URL
                         "http://web.mit.edu/ratings/CodeSafety.html"
                         (ALONG-WITH)))
    (invoke "check-signature" SafetyLabel)
    (match (("load-label")
            (((version "PICS-1.1") *
              (service "http://web.mit.edu/ratings/CodeSafety.html")
              * (ratings (RESTRICT = virus 10)))))
            STATEMENT-LIST)))
  (let
    (EndorsedLabel (invoke "endorse-label" STATEMENT-LIST
                           "mailto:endorsement@mit.edu"
                          "http://web.mit.edu/ratings/endorsement.html"
                           ("http://web.mit.edu/)))
    (invoke "check-signature" EndorsedLabel)
    (match (("check-signature" "load-label")
            (((version "PICS-1.1") *
              (service "http://web.mit.edu/ratings/endorsement.html")
              * (by "mailto:endorsement@mit.edu")
              (ratings (RESTRICT > CodeJudgement 8)))))
            STATEMENT-LIST))))
```

PicsRULZ uses two mandatory extensions, *PicsEndorsement* and *PicsSignature*. The *PicsEndorsement* extension defines a new attribute *endorses* in *serviceinfo*, which contacts the label bureau and requests a PICS label voucher for the author specified in the *endorses* field. *PicsSignature* extension defines a new attribute *check-signature* in *filter* clause, which returns *true* if all the PICS labels specified in its argument have valid signatures. The exact policy for validating a signature is specified in the *PicsSignature* extension.

Profiles-0.92 works in a similar fashion. The policy uses two more REFEREE modules *endorse-label* and *check-signature*. *Endorse-label* takes an auditor, a label bureau, and a rating service as arguments and contacts the label bureau to request labels from the specified rater that vouch for the author of each of the statements on STATEMENT-LIST. *Check-signature* takes a statement-list and validates each PICS label in the statement-list. If a signature is good, it puts its module identifier of the PICS label and returns it. The pattern matcher can verify whether a label is signed by matching "check-signature" in the statement context.

5 REFEREE Reference Implementation

To verify the REFEREE design as described in the thesis, I produced a working REFEREE reference implementation as part of my thesis. I chose Java as my REFEREE underlying execution environment. The implementation work included the core REFEREE primitive data types, REFEREE API, PicsRULZ and Profiles-0.92 interpreters, and a user interface for demonstration purpose. REFEREE was ported to Jigsaw proxy as its host application.

The implementation work turned out to be a simple task. It was a two-month effort with 30 hour input per week. The ease of implementation comes from the simple and elegant design of REFEREE. REFEREE is also efficient; depending on the complexity of the policy, a REFEREE request takes between a quarter of second to half a second to evaluate, excluding the network time to fetch PICS labels. The efficiency shows that Web applications can do trust management without too much of a speed penalty.

Section one introduces the architecture of the Jigsaw proxy, the host application of my reference implementation. Section two describes how REFEREE fits in the Jigsaw proxy. Section three explains the pieces of REFEREE being implemented. Section four provides an execution trace of a sample REFEREE query. Section five discusses several insights and lessons learned from the experience of this implementation.

5.1 Jigsaw Proxy: the Host Application

Jigsaw was originally designed as a Web server, whose purpose was to provide a basis for experimenting new server-side features. Recently Jigsaw introduced a Client API, which manages requests and performs filtering on behalf of a client. The Jigsaw proxy extracts pieces from both the Jigsaw Server and the Client API. The proxy front end, responsible for accepting network requests and managing them as a pool of threads, is taken from the Jigsaw Server front end. The proxy back end, responsible for redirecting requests to Web servers, is taken from the Jigsaw Client API. This section focuses on the Jigsaw proxy back end, in which REFEREE is embedded.

The Jigsaw Client API is very simple: it takes in a *request* (generally a URL) and returns a *reply* (the content of the URL). The API aims to replace the Java standard `java.net.URLConnect` class, with the advantages of being more robust and modular. A simplified architectural figure is shown below.

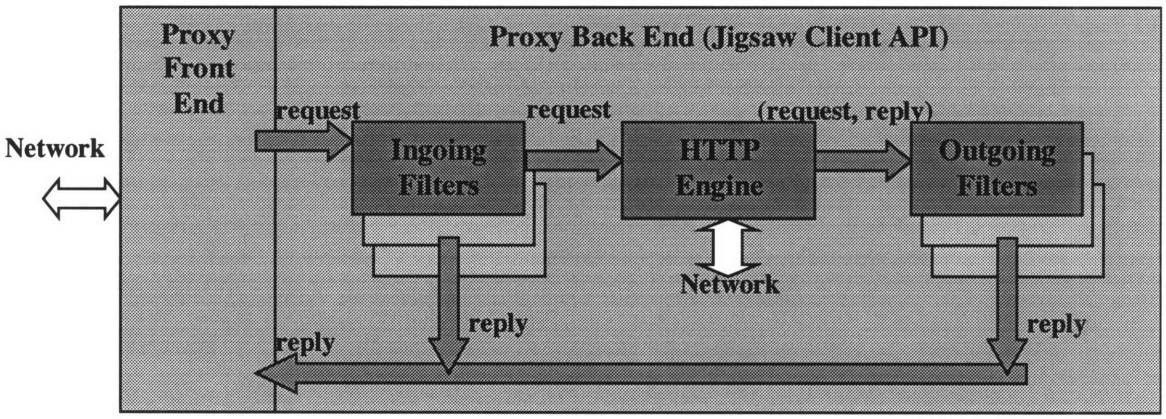

Figure 16 Jigsaw Proxy Architecture

The proxy front end listens to and receives requests from the network. It packages requests from the network and forwards them to the back end, the Client API. Internally, the Client API processes a request object through three boxes in sequence, namely *ingoing filters*, *HTTP engine*, and *outgoing filters*, before a reply object is generated and returned back to the proxy front end.

Ingoing filters

are a set of filters running sequentially. Each *ingoing filter* takes a request as input, and outputs a reply. If the reply is *null*, the request is handed to the next *ingoing filter*. If not, the Client API simply returns the reply without further processing. For example, caching can be implemented as an *ingoing filter*. A cache filter manages a database of cached documents, indexed by URLs. When a request flows into a cache filter, it searches the database with the requested URL. If there is a cache hit, the filter generates a reply object from the database and the Client API terminates and returns that reply object. If there is a cache miss, the filter returns a *null* reply and the Client API continues to the next filter.

HTTP engine

is the engine that fetches information from the network. It makes queries to the appropriate Web server through a network protocol and generates a reply object based on the information received from the network.

Outgoing filters

take both a request and a reply as input, and outputs another reply. If an *outgoing filter* outputs a *null*, the Client API continues to the next filter. If not, the Client API returns the reply object without further processing. For example, authentication can be implemented in an *outgoing filter*. If the input reply is "authentication required", the filter can query the host for password and run the request again, by invoking the Client API with the password.

5.2 REFEREE in the Jigsaw Proxy

REFEREE is implemented as an *ingoing filter* in the Jigsaw proxy. When a request is received, the REFEREE filter constructs an equivalent REFEREE request object, which includes an action name and the URL of interest. The request object is then sent to the REFEREE execution environment for evaluation. If the output of the evaluation is *true*, the filter returns *null*, allowing the request to flow through without interruption. If the output is *unknown* or *false*, the filter returns a default HTML document expressing the request is blocked, along with the justifications returned by REFEREE.

One observation here is that my implementation of the Jigsaw filter has a self-regulating "policy" to respond to the outcome the REFEREE evaluation. That policy is application specific; it is neither controlled nor evaluated by REFEREE, but by the application itself. This observation reinforces the fact that REFEREE is recommendation-based; the burden to enforce the trust management decision is on the application itself. I will discuss more on this aspect in Section 5.5.

Recall from Chapter 3, there are two stages in REFEREE. The bootstrap stage corresponds to the initialization of the REFEREE filter. The query stage corresponds to the invocation of the REFEREE filter. In addition, Jigsaw provides a method (callback) to fetch information from the network. The fetch callback is implemented as the Jigsaw Client API itself, except it does not have the REFEREE filter installed.

5.3 The Scope of the REFEREE Implementation

There are several pieces to the implementation:

REFEREE filter

is a Jigsaw filter that interfaces REFEREE with the Jigsaw proxy. It traps requests in a proxy and hands over to the REFEREE execution environment for evaluation.

REFEREE core module API

is a set of functions calls to initialize and invoke a REFEREE module. The same API is used by the REFEREE filter to invoke the first REFEREE module, and by REFEREE modules to invoke subordinate REFEREE modules.

REFEREE primitive data types

are a set of Java classes shared among REFEREE modules. The primitive data types include *tri-values*, *statement-lists*, and *module-databases*. As in any Java classes, the classes implementing these primitive data types have a set of constructors and a set of methods. For example, *tri-value* class has a set of constructors to create *true*, *false*, and *unknown* objects, and a set of methods to perform *and*, *or*, *not*, *false-if-unknown*, and *true-if-unknown* operations. The *threshold-and* in Profiles-0.92 language is a special operator implemented in the Profiles-0.92 interpreter only.

Profiles-0.92 interpreter

is a trust policy language interpreter implemented as a REFEREE module. It accepts two mandatory arguments, the URL of interests and a list of unconditionally

trusted statements. It also accepts a list of optional arguments. The returned tri-value and statement-list is bound to the returned values of the last policy statement evaluation.

PicsRULZ interpreter

is a trust policy language interpreter implemented as a REFEREE module. It accepts one required input argument, the URL of interest, and no optional arguments. The returned tri-value is either a *true* or a *false*, because PicsRULZ operates on Boolean logic. The returned statement-list is *null*, because PicsRULZ does not return a justification with an answer. Internally, the PicsRULZ interpreter translates the PicsRULZ policy to Profiles-0.92 policy first and then invokes the Profiles-0.92 interpreter with the translated policy.

Label Loader

is a PICS trust protocol implemented as a REFEREE module. There are four required input arguments, a statement-list, a URL of interest, a rating service URL, and a list of label sources. The input statement-list contains a set of cached PICS labels. When there is a cache hit, Label Loader returns the label without fetching it from the network. The URL of interest and the rating service URL specify what PICS labels to fetch. A list of places to find labels includes embedded in a document (by the keyword "EMBEDDED"), via the HTTP header stream (by the keyword "ALONG-WITH"), or from a list of label bureaus.

The returned value is *true* if any label is found, *unknown* if all label bureaus cannot be contact, or *false* if label bureaus can be contacted but no label is returned. The returned statements are parsed PICS labels, which are restructured in a way that are easy for pattern matching and other operations. The exact syntax is in Appendix C. An example is illustrated below, assuming a Profiles-0.92 policy calls Label Loader (local name "load-label") in the following line:

```
(invoke "load-label" STATEMENT-LIST URL
        "http://www.musac.org/v1.0"
        (EMBEDDED "http://www.bureau.com")
```

if Label Loader finds only an embedded PICS label, the returned tri-value is *true*, and the returned statement-list looks like the following:

```
(("load-label")
 (("load-label" "http://www.w3.org/Overview.html" EMBEDDED)
  ((version "PICS-1.1")
   (service "http://www.musac.org/v1.0")
   (by "mailto:mstrauss@research.att.com")
   (original (PICS-1.1 "http://www.musac.org/v1.0"
                  labels by "mailto:mstrauss@research.att.com"
                  ratings (s 1 v 0))
   (ratings (s 1) (v 0)))))
```

As the thesis is written, the implementation of the Load Label module does not have a running PICS protocol. Instead the input of the raw PICS labels are provided by an input stream from the REFEREE filter. The implementation does parse PICS labels and turn them into REFEREE statements.

Label endorser

is a REFEREE module that handles requests for deferral of trust. The module takes three arguments, the name of the auditor, a list of label bureaus, and a statement-list to be endorsed. For each label (represented as a statement) in the statement-list, the module requests additional labels to vouch for the label author (rater) by the name auditor. This module is useful if an application does not know all the raters on the Internet, and instead trust a single auditor who endorses trustworthy raters.

The returned tri-value is *true* if any label in a statement-list is endorsed, *unknown* if the input statement-list contains no PICS label to be endorsed, and *false* if all labels in the statement-list fail to be endorsed. The returned statements simply adds the Label Endorser module's identifier in the statement context and the auditor's name in the statement content on top of the endorsed statement. For example, if an auditor "GoodMouseClicking@w3.org" endorses the rater "mstrauss@research.att.com" from the above statement, the new statement becomes

```
(("endorse-label" "load-label")
 ("mailto:GoodMouseClicking@w3.org"
    ("load-label" "http://www.w3.org/Overview.html" EMBEDDED)
    ((version "PICS-1.1" )
     (service "http://www.musac.org/v1.0")
     (by "mailto:mstrauss@research.att.com")
     (original
        (PICS-1.1 "http://www.musac.org/v1.0"
                  labels
                  by "mailto:mstrauss@research.att.com"
                  ratings (s 1 v 0))
        (ratings (s 1) (v 0)))))
```

As the time the thesis is written, this module is a stub; the REFEREE filter provides an input stream in which a list of trusted auditors are provided explicitly.

5.4 An Execution Trace

This section presents a detail execution trace from my reference REFEREE implementation in the Jigsaw proxy. The REFEREE execution environment is provided with the following bootstrapping information:

identifier	code-fragment	language name
download-applet	<policy to download Java applets>	http://www.w3.org/PICS/Profiles092/
http://www.w3.org/PICS/Profiles092/	<Profiles-0.92 interpreters written in Java>	http://www.javasoft.com/jdk1.1/
load-label	<PICS Label Loader written in Java>	http://www.javasoft.com/jdk1.1/
endorse-label	<Label Endorsement written in Java>	http://www.javasoft.com/jdk1.1/

The bootstrapping statement-list is null, meaning no trusted assertions are unconditionally trusted. *Download-applet* binds to the following Profiles-0.92 policy:

Policy in English

```
Download the code from this URL only if a label from either the HTTP
header stream or from the bureau "bureau.pcworld.com" says it is virus
free (v > 8) according to MIT Code Safety rating, and that rater of the
label is endorsed by the MIT auditor.
```

Profiles-0.92

```
(invoke "load-label" STATEMENT-LIST URL
        "http://web.mit.edu/ratings/CodeSafety.html"
        (ALONG-WITH, "http://bureau.pcworld.com"))
(invoke "endorse-label" STATEMENT-LIST
        "mailto:auditor@mit.edu" ("http://bureau.mit.edu/"))
(false-if-unknown
   (match (("endorse-label" *)
           ("mailto:auditor@mit.edu" *
           ((version PICS-1.1) *
            (service "http://web.mit.edu/ratings/CodeSafety.html") *
            (ratings * (RESTRICT > v 8) * ))))
          STATEMENT-LIST))
```

There are three modules used here, *download-applet* (interpreted by the Profiles-0.92 interpreter), *load-label,* and *endorse-label. Download-applet* module is the top-level module called by the REFEREE filter to interpret the policy shown above. *Load-label* module fetches labels from the network. *Endorse-label* module vouches for labels with raters endorsed by a named auditor. The exact behaviors of the three modules are explained in Section 5.3. Figure 17 shows the order in which the REFEREE modules are invoked.

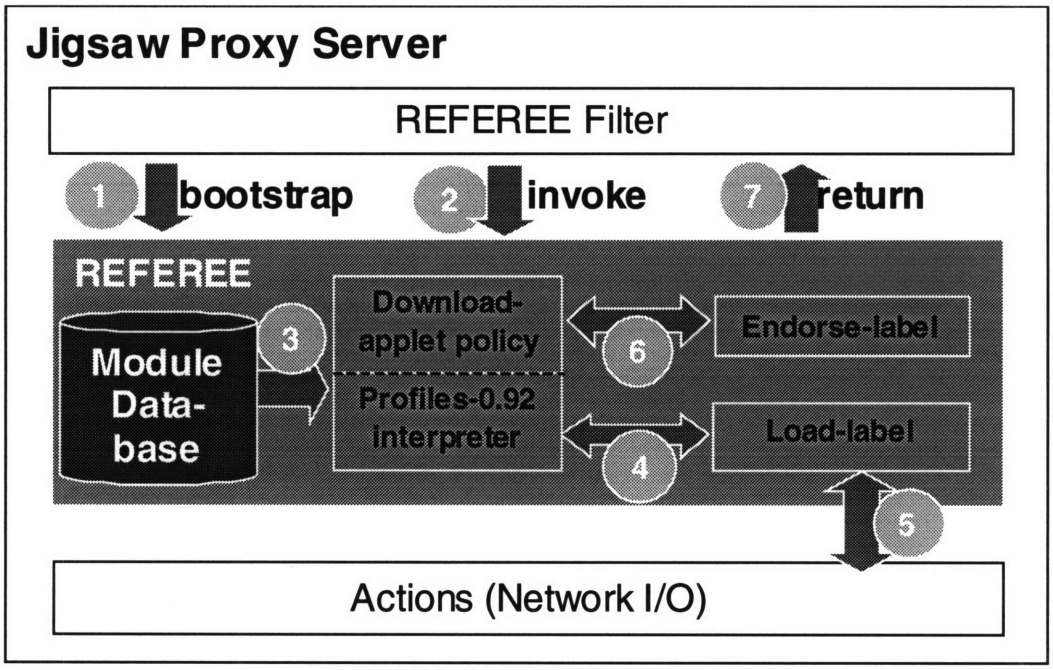

Figure 17 Sample REFEREE Implementation

After the REFEREE filter bootstraps the REFEREE execution environment (step 1), the REFEREE filter is ready to trap requests from the Jigsaw proxy and make queries to the execution environment. When the filter invokes REFEREE with the action name *download-applet* (step 2), REFEREE queries its module database and gets the module containing the pair *download-applet* policy and the Profiles-0.92 interpreter (step 3).

The first line of the *download-applet* policy invokes *load-label* (step 4). Now assume *load-label* actually gets one label from the label bureau "http://www.pcworld.com" (step 5) and returns the following to *download-applet*:

```
      tri-value = true
statement-list = ((()
                  ((version "PICS-1.1")
                   (service "http://web.mit.edu/ratings/CodeSafety.html")
                   (by "mailto:mstrauss@research.att.com")
                   (original (PICS-1.1 ...))
                   (ratings (s 7) (v 9)))))
```

Load-label returns *true*, because a label is found. The statement-list contains a single statement describing the PICS label. The context of the statement is empty, because it is produced by the *load-label* module itself. The caller *download-applet* records this statement by prepending the name of the called module, "load-label," onto the context of the statement:

```
statement-list = ((("load-label")
                  ((version "PICS-1.1")
                   (service "http://web.mit.edu/ratings/CodeSafety.html")
                   (by "mailto:mstrauss@research.att.com")
                   (original (PICS-1.1 ...))
                   (ratings (s 7) (v 9)))))
```

onto its local copy of the statement-list.

Now *download-applet* proceeds to the second line, invoking the module *endorse-label* to check for an endorsement (step 6). Assuming the endorsing label is found, *endorse-label* returns a statement-list that gets "endorse-label" prepended to the context, resulting in the following:

```
      tri-value = true
statement-list = ((("endorse-label" "load-label")
                  (("mailto:auditor@mit.edu")
                   ((version "PICS-1.1")
                    (service "http://web.mit.edu/ratings/CodeSafety.html")
                    (by "mailto:mstrauss@research.att.com")
                    (original (PICS-1.1 ...))
                    (ratings (s 7) (v 9))))))
```

Again, the string "endorse-label" is added to the context of this statement to indicate that the rater "mailto:mstrauss@research.att.com" is approved by *endorse-label* policy. The passed content is wrapped in an expression containing "mailto:auditor@mit.edu", the name of the auditor.

Finally, *download-applet* proceeds to the last line to check ratings. The match looks for a context with "endorse-label"—if it is missing, the match fails. Because the match succeeds, *download-applet* returns to the application a tri-value of *true* and a statement-list of the statements produced by the matcher:

```
      tri-value = true
statement-list = ((("endorse-label" "load-label")
                  (("mailto:auditor@mit.edu")
                   ((version "PICS-1.1")
                    (service "http://web.mit.edu/ratings/CodeSafety.html")
                    (by "mailto:mstrauss@research.att.com")
                    (original (PICS-1.1 ...))
                    (ratings (s 7) (v 9))))))
```

The returned values say the *download-applet* action should be taken (tri-value is *true*), and it is justified by the endorsed PICS label in the statement-list. They are returned to the REFEREE filter (step 7), and the filter returns a *null* reply object to the proxy Client API. The null reply allows the request to resume processing in the Jigsaw proxy.

5.5 Discussions

The REFEREE implementation in the Jigsaw proxy provides many insights on how a trust management system should work in a real-world application. This section attempts to address some of the concerns raised during the implementation, and explain how my particular implementation deals with them.

The first concern is the order in which REFEREE is placed with respect to other tasks in a host application that are concurrently affecting the behavior of the application. Caching is such a conservative task in the Jigsaw proxy, where the order to do trust management and caching matters. I place the REFEREE filter in the highest precedence, so it always gets evaluated. It is considered the most conservative approach, because it would accurately observe time-dependent trust elements, such as expired or revoked certificates, and make correct trust decisions based on them. However, if performance is more critical than accuracy, an application should place the caching filter in front of the REFEREE filter. Determining how the trust management task interacts with other processes in a host application is in a way another "trust policy", and it is outside the scope in which REFEREE can evaluate.

The second concern is whether actions issued during the evaluation of trust management are subject to the same trust management evaluation. For example in the Jigsaw proxy, *download-applet* requires *Label Loader* to call the Jigsaw Client API and fetch PICS labels from the network. The act of fetching PICS labels itself may be considered as a trust management problem and be subjected to a label-fetching trust management policy. My current implementation does not invoke another level of trust management during a trust management evaluation. I reject this idea for two reasons. First, it may introduce deadlock if the label-fetching policy in turn requests the same labels before the label can be fetched. The same Label Loader would be called recursively without making any progress. Second, I treat the action of fetching PICS labels as a trust protocol that is safe, secure, without any judgement of trust, therefore the action needs not be subjected to a trust management decision.

The third concern is whether REFEREE should introduce an explicit caching mechanism for performance reason. Currently REFEREE does not have one, and my implementation has no mechanism explicitly for caching purpose. However, my implementation transparently inherits the benefit of Jigsaw's internal caching mechanism (caching filter). When *Label Loader* needs to fetch labels from the network, it calls the Jigsaw Client API, where the cache filter is activated. The subsequent call to get the same label will be caught by the caching filter, and hence *Label Loader* gets caching for free. The observation implies that caching is supposed to be transparent from the rest of the processes in the application, and REFEREE needs not implement an explicit caching mechanism.

The fourth concern is whether REFEREE can be application independent as promised. As demonstrated in this implementation, the only two pieces that are "Jigsaw-centric" are the Jigsaw filter and the network fetcher. The Jigsaw filter traps requests from its host and bootstraps REFEREE. The network fetcher fetches information from the network, which are already in place for most network applications. Both of them are considered minimal for an application to do trust management. The rest of the code can be ported to other applications without any modification.

The fifth concern is whether REFEREE introduces disastrous performance hit for doing trust management. My implementation takes less than half a second to evaluate of the sample policy in Section 5.4, excluding any network time (my implementation supplies all the network information through a fixed input stream). This observation implies that the bottleneck to do trust management will not be the invocation of REFEREE modules, or evaluation of trust policies. Rather, the bottleneck will be the use of network, where fetching labels from the Web may incur long delays, or the use of cryptography, where validating digital signatures may take large CPU cycles. They are however, the unavoidable steps to make any trust decisions, but the overhead of REFEREE is minimal.

6 Conclusion

My thesis identifies the trust management problems in the context of the World Wide Web and provides a two-part solution: REFEREE as the general-purpose execution environment and PicsRULZ and Profiles-0.92 as the policy languages. They utilize the existing trust protocols and metadata formats, and together, they form a complete trust management infrastructure in which trust is exchanged and established among mutually untrusting parties in an untrusted information infrastructure.

My thesis has four contributions to the area of trust management:

- identify the concept of the *trust management infrastructure*, with the four basic building blocks in the infrastructure.
- study current protocols and systems involving trust and identify their strengths and weaknesses.
- propose a two-part solution: REFEREE as a generic execution environment, and Profiles-0.92 as a flexible trust policy language.
- implement reference versions of REFEREE and Profiles-0.92 and prove that the concept of a generic trust management infrastructure is a realistic goal.

I do not claim the work on REFEREE and trust management is definitive or conclusive in its current state, but rather that it is a step forward in the understanding of the intricacies of trust. Of course, more work is needed. In particular, we need network experts to build robust and yet more efficient metadata formats and trust protocols. We need language experts to define simple and yet expressive trust policy languages. We need system experts to structure secure and yet dynamic execution environment. We also need user interface experts to deliver a user-friendly and yet feature-rich user interface to take advantage of a sophisticated trust management infrastructure beneath.

More challenges are ahead of us. Chin up!

Appendices

Appendix A. Modified BNF for PicsRULZ Policy Language

```
rule            :: '(' 'PicsRule-' verMajor '.' verMinor rule-body ')'
verMajor        :: integer
verMinor        :: integer
rule-body       :: '(' rule-clauses ')'
rule-clauses    :: rule-clause+
rule-clause     :: filter-clause | fail-clause | pass-clause |
                   name-clause | source-clause | service-clause |
                   opt-ext-clause | req-ext-clause |
                   extension-clause
filter-clause   :: 'Filter' '(' attrvalpair+ ')'
fail-clause     :: 'failURL' '(' attrvalpair+ ')'
url-list        :: quotedURL+
pass-clause     :: 'passURL' '(' attrvalpair+ ')'
name-clause     :: 'name' '(' attrvalpair+ ')'
source-clause   :: 'source' '(' attrvalpair+ ')'
service-clause  :: 'serviceinfo' '(' attrvalpair+ ')'
opt-ext-clause  :: 'optextension' '(' attrvalpair+ ')'
req-ext-clause  :: 'reqextension' '(' attrvalpair+ ')'
ext-clause      :: extension-clause-name '(' attrvalpair+ ')'
attrvalpair     :: attribute whitespace value | primaryvalue
attribute       :: alphanumstr
value           :: quotedstring | '(' attrvalpair '(' whitespace
                   attrvalpair)* ')'
primaryvalue    :: quotedstring+ | '(' attrvalpair+ ')'
quotedstring    :: ('"' notquotechars '"') | ("'" notquotechars "'")
alphanumchar    :: alphanum+
whitespace      :: ' ' | '\t' | '\r' | '\n'
alphanum        :: '0' - '9' | 'A' - 'Z' | 'a' - 'z'
notquotechars   :: any ASCII characters between 32-127 except ' and "
comment         :: '{' comment-text* '}'
comment-text    :: any octets except '}'
PermissionExp   :: "Unless-Prohibited" | expression
ProhibitionExp  :: expression
expression      :: simple-expression | or-expression | and-expression
simple-exp      :: '(' service '.' category op constant ')'
service         :: any shortname defined in a serviceinfo clause
                   within this rule
category        :: any transmit-name for a category defined by
                   the rating-system referred to by the matching system
op              :: '>' | '<' | '=' | '!=' | '>=' | '=>' | '<=' | '=<' |
                   'all-equal' | 'none-equal' | 'includes'
constant        :: [sign] alphanumchar ['.' alphanumchar]
or-expression   :: '(' expression or expression [or expression]+ ')'
or              :: 'or' | '||'
and-expression  :: '(' expression and expression [and expression]+ ')'
and             :: 'and' | '&&'
sign            :: '-'
```

Appendix B. Modified BNF for Profiles-0.92 Policy Language

```
policy          :: rule+
rule            :: let-rule | combine-rule | invoke-rule |
                   install-rule | project-rule | match-rule |
                   url-match-rule
let-rule        :: '(' 'let' '(' let-binding+ ')' rule+ ')'
let-binding     :: '(' variable-name let-expr ')'
variable-name   :: symbol
let-expr        :: rule | ''
combine-rule    :: unary-rule | multi-rule | threshold-rule
unary-rule      :: '(' unary-op rule ')'
unary-op        :: 'not' | 'true-if-unknown' | 'false-if-unknown'
multi-rule      :: '(' multi-op rule* ')'
multi-op        :: 'and' | 'or'
threshold-rule  :: '(' 'threshold-and' threshold-val rule* ')'
threshold-val   :: number
invoke-rule     :: '(' 'invoke' policy-name statement-list
                   optional-arg* ')'
policy-name     :: quoted-name
statement-list  :: '(' statement* ')'
statement       :: '(' context content ')'
context         :: s-expression
content         :: s-expression
optional-arg    :: s-expression
install-rule    :: install-policy | install-interp
install-policy  :: '(' 'install-policy statement-list ')'
install-interp  :: '(' 'install-interpreter' statement-list ')'
project-rule    :: '(' 'tri-value' rule ')' |
                   '(' 'statement-list' rule ')'
match-rule      :: '(' 'match' pattern rule ')'
pattern         :: '*' | '+' | '.' | string-literal | symbol-literal |
                   restriction | '(' pattern* ')' | '\' string-literal |
                   ',' string-literal
restriction     :: '(' 'RESTRICT' restriction-op transmit-name
                   value ')'
restriction-op  :: '<' | '>' | '=' | '<=' | '>=' | '<>' |
                   '<!' | '>!' | '=!' | '<=!' | '>=!' | '<>!'
transmit-name   :: as defined in [PICS97a]
url-match-rule  :: '(' 'url-match' variable-name string-literal+ ')'
```

Appendix C. Modified BNF for the Returned Statement-List of Label Loader

```
returned-stmt   :: '(' content* ')'
content         ::  '(' header version service poption* ratings ')'
header          :: '(' '"label-loader"' quotedURL label-source ')'
label-source    :: bureau | 'EMBEDDED' | 'ALONG-WITH'
bureau          :: quotedURL
version         :: '(' 'version' '"PICS-1.1"' ')'
service         :: '(' 'service' quotedURL ')'
poption         :: '(' option ')'
option          :: 'by' quotedname | 'gen' Boolean |
                   'for' quotedURL | 'on' quoted-ISO-date |
                   'signature-rsa-md5' base64-string |
                   'exp' quoted-ISO-date | 'at' quoted-ISO-date |
                   'md5' base64-string | 'comment' quotedname |
                   'full' quotedURL |  'original' quotedname |
                   'extension' '(' mand/opt quotedURL data* ')'
ratings         :: '(' 'ratings' rating* ')'
rating          :: '(' transmit-name number ')' |
                   '(' transmit-name '(' multi-value* ')' ')'
transmit-name   :: as defined in [PICS97a]
alphanumpm      :: 'A' | ... | 'Z' | 'a' | ... | 'z' |
                   '0' | ... | '9' | '+' | '-'
urlchar         :: alphanumpm | '.' | '$' | ',' | ';' | ':' | '&' |
                   '=' | '?' | '!' | '*' | '~' | '@' | '#' | '_' |
                   '%' hex hex
```

References

[BCKLMRS] Brezin, J., Chu, Y., Khare R., LaMacchia, B., Miller, J., Resnick, P., Strauss, M. (1996), "REFEREE Version 1.4d: Rule-controlled Environment for Evaluation of Rules, and Everything Else," *Working draft.*

[BFL96] Blaze, M., Feigenbaum, J., and Lacy, J. (1996), "Decentralized Trust Management," *Proceedings of the 17th Symposium on Security and Privacy,* IEEE Computer Society, Los Alamitos, 1996, pp. 164-173.

[BFRS97] Blaze, M., Feigenbaum, J., Resnick, P., and Strauss, M. (1997), "Managing Trust in an Information-Labeling System," to appear in *European Transactions on Telecommunications.* (Special issue of selected papers from the 1996 Amalfi Conference on Secure Communication in Networks.)

[CCITT88a] CCITT (1988), Recommendation X.500 (ISO 9594): The Directory—Overview of Concepts, Models and Services.

[CCITT88b] CCITT (1988), Recommendation X.509: The Directory--Authentication Framework.

[CFLRS96] Chu, Y., Feigenbaum, J., LaMacchia, B., Resnick, P., and Strauss, M. (1997), "REFEREE: Trust Management for Web Application," *Proceedings of the 6th International World Wide Web Conference Proceedings,* International World Wide Web Conference Committee, 1997, pp. 227-238.

[DLLC97] DesAutels, P., Lipp, P., LaMacchia, B., and Chu, Y. (1997), "DSig 1.0 Signature Labels – Using PICS 1.1 Labels for Digital Signature,'*W3C Working Draft,* June 5, 1997.

[EFRT97] Ellison, C., Frantz, B., Rivest., and Thomas, B. (1997) "SPKI—Simple Public Key Certificate," *Internet Draft,* April 6, 1997.

[FL97] Feigenbaum, J., and Lee, P. (1997), "Trust Management and Proof-Carrying Code in Secure Mobile-Code Applications," *PDARPA Workshop on Foundations for Secure Mobile Code,* March 26-28, 1997.

[MS96] Microsoft Corporation (1996), "Proposal for Authenticating Code Via the Internet." *Revision 1.1*, available from http://www.microsoft.com/workshop/prog/security/authcode/authcode.htm .

[NCSA95] National Security Computer Association (1995), "Microsoft Word Document Macro Virus," *Hard-Copy, Journal of the Chicago Computer Society, Volume 11, Number 10, October 1995, pp. 39-40,* available from http://www.ccs.org/hc/9510/ncsa.html.

[Necula97] Necula, G. (1997), "Proof-Carrying Code," to appear in *Proceedings of the 1997 ACM Symposium on Principle of Programming Languages.*

[PICS97a] "Rating Services and Rating Systems and Their Machine-Readable Descriptions Version 1.1," *W3C Recommendation.*

[PICS97b] "PICS Label Distribution Label Syntax and Communication Protocols Version 1.1," *W3C Recommendation.*

[PICS97c] "PICS Profile Language Working Group - PicsRULZ,'*W3C PICS Working Group draft*, available from http://www1.raleigh.ibm.com/pics/ProfilesWG.html.

[RL96] Rivest, R. and Lampson, B. (1996), "SDSI—a Simple Distributed Security Infrastructure Version 1.1," *draft*, available from http://theory.lcs.mit.edu/~cis/sdsi.html.

[RSA97] RSA Laboratories (1997), "PKCS #7: Cryptographic Message Syntax Standard Version 1.6," *RSA Standards.*