A TRUST-BASED MODEL FOR INFORMATION INTEGRITY IN OPEN SYSTEMS[!]

Yanjun Zuo[1] and Brajendra Panda[2]
[1]*Department of Information Systems and Business Education, University of North Dakota, Grand Forks, ND, USA;* [2]*Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, USA*

Abstract: While it is difficult to apply conventional security services to a system without a central authority, trust management offers a solution for information assurance in such a system. In this paper, we have developed a policy-oriented decision model based on object trust management to assist users in selecting reliable and secure information in an open system. In the proposed model, an object represents a topic or issue under discussion, and it may have multiple versions, each of which represents a subject's opinion towards the characteristics of that object. The developed trust-based decision model assists a user to select one object version with desired level of quality and security features from available versions of a given object. The model balances both positive and negative aspects of an object version, and an evaluator can explicitly specify, in form of a policy specification, which features of an object version are not acceptable and which features are favorable. A high-level policy language, called *Selector*, expresses the policy specification in an unambiguous way. *Selector* consists of primary and residual policy statements. It supports recursive function calls, and the invoked external functions are defined separately from the language itself. The proposed decision model doesn't guarantee to select the "best" version for a given object. Rather it ensures that the selected version meets a user's requirement for information integrity.

Key words: trust decision model; information integrity; information security policies; policy language; trustworthy computation

[!] This work was supported in part by US AFOSR under grant FA9550-04-1-0429 and was performed when the first author was with the University of Arkansas.

1. INTRODUCTION

Information integrity has a wide scope and it primarily used to refer to a set of mechanisms to protect information from unauthorized modifications during the information transmission or in storage. In this paper, information integrity focuses on evaluating the quality and security features of a given piece of information. It contains a set of methods for an evaluator to select external information with the required level of quality and security in an open environment. An open system is a general term and, in this paper, it represents a decentralized system organized by a set of loosely coupled computer systems without a single administrative authority. Examples of open systems include various virtual organizations such as Grid systems, Peer-to-peer systems, and virtual communities. Ensuring the security and quality features of external information is crucial for the participants of an open system to confidently share information. But the conventional security and information assurance mechanisms don't scale well while being applied to those open systems. They have been developed based on a closed-world assumption where the users are known in advance. This assumption is no longer valid for an open system. Rather, trust management helps eliminate the scalability limitation of traditional security models. Existing research on trust management focuses on subject trust, however, e.g., how the trustworthiness of a subject is evaluated and how access control is granted to a subject based on its attributes and/or properties. Research on object trust has not received much attention.

In our discussion, a *subject* represents an independent entity in an open system, which produces and consumes information. A piece of information expresses a topic or issue in discussion. The term *object* is used to denote such a topic or issue. This notation (object) is chosen because it is frequently used together with the term, subject. An object has a value or a set of values, called object value(s), representing the inherent features(s) of the object. For instance, if the current economic growth is considered as an object, then its object value is a real number representing how fast the economy is growing. The object values expressed by different subjects for a given object could be different. It is very likely that different subjects have different views on a given issue or topic. For example, different groups of economists may have used different analytical tools and collected different sets of data to calculate the economic growth rate. Hence, they have different opinions on this value.

The term *object version* is used to represent such an opinion that a subject has on the object value(s) for the given object. In addition, the owner of an object version also supplies its confidence in the proposed object value(s), which is expressed as the trust that the owner places on the proposed object value(s).

Information processing is accumulative and recursive, e.g., some information is formed by using others as its components. For instance, a public key encryption algorithm (e.g., RSA) uses those methods for large primary number generation and testing, key distribution, and one-way function (e.g., modular operation) as its building blocks. In component-based software development, e.g., Java Beans and Microsoft COM, a software program is constructed by using various pre-developed modules, library functions, and methods. In business world, the Dow Jones Industrial Average summarizes 30 stock prices in average and divides it by a constant, called "divisor". Information derivation is a major form of information processing in a data intensive system for science and commerce. In domains as diverse as global climate change, high-energy physics, and computational genomics, science is becoming increasingly dependent on the generation and reuse of massive amounts of data, a trend sometimes known as data-intensive science.

Our model is applied to an open system, where information derivation enables the system to keep track of the components of an object version, i.e., how the object version has been formed and which components are used. Then that information is helpful for a user to evaluate the trustworthiness of the object version in term of its quality and security.

In [1] the authors proposed a standard format to represent different versions of a given object and the component information for each version. Furthermore, they developed a method to allow an evaluator to measure the trustworthiness of an object version based on the trust values of its components and the composing functions used to form the object version. For simplicity, an object version is specified in the following format:

owner → {object, object value(s), trust value, components,

composing functions}

An object can have multiple versions as provided by different subjects. To distinguish among available versions of an object, say O, the symbol $V^{(O)}_i$ is used to denote the i^th^ version of O. O is called the target of object of $V^{(O)}_i$. Given multiple versions of an object, a user may want to select one version, if any, which satisfies its requirements for quality and/or security. This process is called "trust decision", which is an important part of object trust management. Existing virtual organizations provide only preliminary approaches for selecting trusted information in term of its quality and security. For instance, in Peer-to-peer systems, a peer assesses a given piece of information based on the reputation and trustworthiness of the owner of the information. But that is not a reliable way to evaluate the quality and security features of the information itself. In this paper, a trust-based decision model has been developed. The selection criteria are defined based on both the intrinsic and extrinsic features of the given information, and those features provide an evaluator more insights into the inherent trust characteristics of the information. A policy specification can then be defined based on this trust model, and a high-level policy language is applied to formally express the policy specification.

2. RELATED WORK

Existing research on trust management includes trust modeling [2, 3, and 4], automatic trust negotiation [5, 6, and 7], reputation based trust management [8, 9, and 10], among others. Examples of trust management systems include PGP [11], X.509 [12], PolicyMaker [13], KeyNote [14], Referee [15], etc. Our model is different from the previous work in that our approach concentrates on evaluating the trustworthiness of a given object version (or a piece of information), while the existing models focus on trust at subject level. Studying the information quality and security at object level gives a user higher confidence to use a piece of information since that information has been directly assessed instead of relying on the information's extrinsic attributes such as its owner's reputation. Making a decision merely based on a subject's trustworthiness is not always reliable. We know that even the most honest people make mistakes. It is advantageous to assess the intrinsic features of external information and perform trust evaluation at object level. Then the information can be consequently selected based on its trust features.

Related work on policy languages includes [16, 17, and 18] (to cite a few). In [16] McDaniel has discussed in detail execution conditions in order to determine if a policy should be applied. *Ponder*, a policy language as proposed in [17], consists of a set of statements that define a choice in the behavior of a system. The language itself is declarative and object oriented. In [18], *Rei*, another policy language, was introduced. The core of that policy language is policy objects, which describe the concepts of rights, prohibitions, obligations, and dispensations. The "*has*" construct as defined in *Rei* represents the possession of a policy object by a subject.

Selector, the high-level policy language presented in this paper, is simple and flexible. An instance of *Selector* is composed of a set of policy statements. A *select* or *deny* primary statement immediately selects or denies an object version, which offers some features that the user has strong likeness for or can't tolerate, respectively. A *warning* or *rewarding* policy statement allows the accumulations of negative or positive effects of a given object version. Then the implied statements are applied to test if the accumulated effects are significant enough to make a decision.

Like *Ponder* and *Rei*, *Selector* supports positive and negative rules as well as recursive external function calls. But *Selector* is designed specifically for expressing a policy specification to select external information by evaluating the intrinsic and extrinsic feature of the information. This feature makes it different from both *Ponder* and *Rei*, which focus on user aspect and specify policies for management and security of distribute systems. For instance, *Ponder* includes authorization, filter, refrain and delegation polices for specifying access control and obligation policies to specify management actions, and supports a common means of specifying enterprise-wide security policy [17]. Key concepts of *Ponder* include domains, roles, and relationships, which support it as an object-oriented policy language. *Rei* handles authorizations, prohibitions, obligations and dispensation policy rules and allows policies to be split into actions, constraints and policy objects. Hence both languages specify rules to describe allowed actions for subjects and *Selector* specifies favorable and prohibitive features of objects. In addition, *Rei* defines actions and policy objects separately and allows them to be linked dynamically to subjects. *Selector* uses domain dependent function blocks in rule expressions. The feature of domain dependency increases implementation efficiency, as compared with *Rei*, at the expense of extensibility and portability.

3. THE TRUST-BASED DECISION MODEL AND POLICY SPECIFICATION

3.1 Methodologies

The goal of the trust decision process is to select one version, if any found appropriate, from available versions of a given object. Making a trust selection relies on evaluating trust features of those available object versions. The trust features of an object version are quantitatively expressed by its values for a set of trust-related attributes of the target object. The concept of trust-related attribute is defined as below.

Definition 1: A *trust-related attribute* of an object refers to the object's intrinsic or extrinsic attribute, whose value, given a version of the object, describes the quality and/or security features of the object version and hence can help an evaluator assess the trustworthiness of that object version.

Figure 1 describes object O, its three trust-related attributes, and the values of those attributes given O's three versions. The attribute, "possibility of viral infection", describes one security feature of O's versions. Given a version of O, its value for this attribute helps an evaluator measure how much to trust this object version in term of its safety to execute, i.e., free of viral infection. Another attribute, "correctness of the algorithms", concerns the quality feature of a version of O. The value for this attribute, given an object version, helps the evaluator decide how much the quality (correctness) of the object version should be trusted in term of the algorithms it used to solve a problem. The third attribute, "the owner's reputation", indicates the reputation of the object version's owner in supplying information. An object version's value for this attribute, i.e., its owner's reputation, provides the evaluator important information in measuring the quality and/or security feature of the object version.

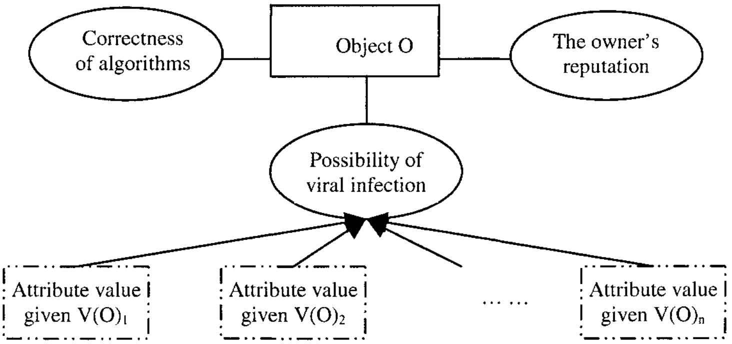
Correctness of algorithms
Object O
The owner's reputation
Possibility of viral infection
Attribute value given $V(O)_1$
Attribute value given $V(O)_2$
......
Attribute value given $V(O)_n$

*Figure 1.*Object O, its Attributes, and the attributes' values given O's three Versions

Some trust-related attributes are considered as positive semantically in the sense that users want to see higher values for them. For instance, the reputation of the owner of a given object version is a positive trust attribute. In contrast, some attributes are considered negative, and users want to see lower values for those attributes. For instance, the possibility of virus infection for an object version describes a negative feature for that version. Users want to see this possibility as very low. When an object version has a high value for a negative attribute, and a user can't tolerate that feature expressed by this negative attribute value, then the object version must be rejected.

Definition 2: A *dominating negative attribute value* of an object version refers to such a value of the object version for a negative attribute that the feature represented by this value is so "negative" that a system can not accept the object version based on this feature.

The corresponding attribute is called a *dominating negative attribute* of the target object.

In a trust selection process, a user first identifies a set of *dominating negative attributes* and specifies a set of testing conditions such that if any object version possesses a value for one of the dominating attributes and that value is beyond a threshold (i.e., the object version has a very negative feature and can't be accepted by the user), the object version must be denied.

On the other hand, an object may have a positive attribute which is so attractive. If a version of that object has a value for such a favorable attribute, and that value is good enough (or beyond a threshold), then the object version can be selected.

Definition 3: A *desired positive attribute value* of an object version is such a value of the object version for a positive attribute that the feature represented by the value is highly "favorable." Moreover, the object version can be accepted by a system if the object version does not possess any dominating negative attribute values.

The corresponding favorable attribute is called a *desired positive attribute* of the target object.

As an object version can have complicated features, it is difficult to draw a clear line between a "good" version and a "bad" version. An object version may have some positive trust features; but these merits are not good enough for a user to make a "select" decision. On the other hand, a version may have some negative aspects but those unfavorable features are not severe enough for the user to make a "deny" decision. Hence the proposed trust selection process uses a scoring system to allow the values for those negative and positive features to be quantitatively accumulated. Then a decision may be made based on those accumulated negative or positive features. A policy specification can then be developed for selecting an object version, if any found appropriate, which satisfies the user's requirements for information quality and security.

Definition 4: A *policy specification* consists of a set of trust based security rules, called *policy rules*, expressed in a natural language specifying high-level descriptions of what features of an object version can not be tolerated and thus that object version must be denied as well as what features are favorable and thus that object version can be accepted.

There are four types of policy rules in a policy specification as discussed below:

(1) The first type is to specify dominating negative attribute(s) of an external object and the corresponding testing conditions.

(2) The second type is to specify desired positive attribute(s) of an external object and the corresponding testing conditions.

(3) The third type is to specify either positive or negative attribute(s), for which one object version has a value for that attribute but that value is not significant enough (for an evaluator to make a "select" or "deny" decision). However, those positive or negative features should be accumulated accordingly for the object version. Two variables, *Number of Rewards* and *Number of Warnings*, are

introduced to maintain the accumulated utilities for the positive and negative features of an object version respectively.

(4) The fourth type is to verify whether the accumulated positive and negative utilities for an object version under evaluation enable the system to make a decision after balancing their overall effects.

A policy rule of type (1), (2), or (3) is also called a *primary rule.* More specifically, a rule of type (1) is called a *primary negative rule*, and a rule of type (2) is called a *primary positive rule.* A rule of type (3) is called an *accumulating rule.* A rule of type (4) is called an *implied rule.* An implied rule takes the pair of accumulated positive and negative features of a given object version as input and produces one value from set {*deny, select, indecisive*}.

All the primary policy rules are evaluated and enforced in a pre-defined order. Primary negative rules, if any, can not appear after any other types of primary rules. In other words, a policy specification must start with a set of primary negative rules, if any. Those primary negative rules specify dominating negative attributes of an object and their testing conditions. If an object version has one of those values as tested true by such a condition (i.e., it has a negative feature that can not be accepted by the system), that object version must be denied. If an object version does not possess any of the negative features as specified by a primary negative rule, it is tested based on other primary positive rules and accumulating rules. Finally, if all the versions of the given object have been evaluated and no version has been selected, then a *residual rule* (as will be discussed in Section 4.3) is applied and a version may be selected for that object.

3.2 An Example

A policy specification example has been given in Figure 2, where system S evaluates object version $V^{(O)}_i$ to determine if $V^{(O)}_i$ satisfies its requirements for information quality and security. The policy specification starts with a primary negative rule, which indicates that if $V^{(O)}_i$ is detected as infected by viruses, then S rejects $V^{(O)}_i$ immediately and the following rules of the policy specification will not be evaluated. For this rule, the dominating negative attribute is "infection by viruses", and the corresponding testing function is "verify if a given object version has been affected by viruses". If $V^{(O)}_i$ is tested as "false", i.e., $V^{(O)}_i$ has not been detected as affected by a virus, then the evaluation continues and the second policy rule is evaluated for $V^{(O)}_i$. The second policy rule is a primary positive rule. According to this rule, S can select $V^{(O)}_i$ if (1) $V^{(O)}_i$'s components are publicly known as "recommended object versions" (hence publicly known as trustworthy). It is assumed that all recommended object versions based on user evaluations are maintained in a list, called *RecommendedObjectVersionList* and this list is accessible to all participant in a virtual organization, and (2) the composing functions used to form $V^{(O)}_i$ have been verified as correct and appropriate. For this rule, "component correctness" and "composing logic correctness" are two desired positive attributes. The first is checked by its corresponding testing condition to determine if $V^{(O)}_i$'s components are members of *RecommendedOjbectVersionList.* The second is to determine (by site domain experts or external service providers) if the composing functions are correct and appropriately used. If the answers to both of the two testing conditions are "yes", then $V^{(O)}_i$ can be selected and the decision process for the target object, O, is completed. If the answer to at least one of the testing conditions is "no", the evaluation process continues and the next rule in the policy specification is evaluated for $V^{(O)}_i$. The third rule is also a primary positive rule and can be interpreted in a similar way. The fourth rule is an accumulating rule. It indicates that for a positive attribute, "membership of the owner of a given object version to *GoodContactList*", if $V^{(O)}_i$ satisfies the corresponding testing condition, two units of rewards can be accumulated for $V^{(O)}_i$, i.e., *Number of Rewards* is incremented by two. *Good ContactList* is maintained locally by the evaluator and contains all other subjects, which the evaluator has contacted before and whose performance were satisfied. Similar explanations can be applied to the remaining rules.

After all the primary rules have been evaluated, the accumulated *Number of Rewards* and *Number of Warnings* for $V^{(O)}_i$ are used to determine if $V^{(O)}_i$ should be selected, denied, or indecisive based on the two implied rules.

The policy specification (to evaluate object version $V^{(O)}_i$):

(Primary rules):

1. Deny $V^{(O)}_i$ if it has been affected by viruses;
2. Select $V^{(O)}_i$ if $V^{(O)}_i$'s components are publicly known as trustworthy and the composing functions used to form $V^{(O)}_i$ are verified as correct;
3. Select $V^{(O)}_i$ if the overall trust value of $V^{(O)}_i$ (the combination of its primary and secondary trust values) is greater than 0.95;
4. Give $V^{(O)}_i$ two unit of rewards if the owner of $V^{(O)}_i$ had been in transaction with S and its performance was satisfied;
5. Give $V^{(O)}_i$ two units of warnings if the owner of any component of $V^{(O)}_i$ has reputation value less than 0.4;
6. Give $V^{(O)}_i$ two unit of rewards if the owner of any component of $V^{(O)}_i$ is S's business partner.
7. Give $V^{(O)}_i$ three units of warnings if the program expressed by $V^{(O)}_i$ has been tested with memory leakage while being executed.

Figure 2. Example of a Policy Specification

In the evaluation process of an object version, if a "select" decision for that version is made, the process of evaluating the target object is completed. If a "deny" decision for an object version is made, the evaluation process only for that version is completed. In both cases, the remaining policy rules for that version are not evaluated or applied.

4. *SELECTOR* – THE POLICY LANGUAGE

A policy specification is defined in a natural language, e.g., English. Rules expressed in a natural language could be ambiguous. In computer science literature, a policy language with well-formatted syntax is often used to express a policy specification. In this paper, a policy language, called *Selector*, has been developed to express a policy specification.

Selector is a high-level policy language. According to Bishop [20], a high-level policy language expresses policy constraints on entities using abstractions without specifying the implementation issues. Translating a policy specification to an instance of a policy language is conducted manually.

Selector consists of a set of policy statements to express the corresponding policy specification rules. The term "statement" is used for *Selector* in order to distinguish it from the term "rule" for a policy specification. More specifically, *Selector* is composed of a "primary statement list" and a "residual statement list". The residual statements help an evaluator select a version if all the primary statements have been evaluated based on the available versions of a given object and no version has been selected.

```
Selector            ::= primary_statements  implied_statements residual statements
primary_statements ::= primary_statement primary_statements | ε
 primary_statement ::=  select_statement | deny_statement |
```

Figure 3. High Level Structure of Syntax of *Selector*

The high-level structure of *Selector* is expressed in Figure 3. The syntax of both primary statements and implied statements of *Selector* is introduced in the following sections.

4.1 Primary Statements

A primary policy statement has the following format:

Action $\leftarrow$ *(LoopControl:)? Fc(*C_1*,* C_2*, ...,* C_n*)*

The terms for the above statement are explained below:

(1) *Action* $\in$ {Select, Deny, [Warnings *x*], [Rewards *y*]} and each element is a self-explanatory function identifier. There are four types of primary statements: (a) *Select statement*: the *Action* is *Select*. Such a statement expresses a primary positive rule in the corresponding policy specification; (b) *Deny statement*: the *Action* is *Deny*. Such a statement expresses a primary negative rule in the policy specification; (c) *Warning statement*: the *Action* is [Warnings *x*]. Such a statement expresses an accumulating rule in the policy specification to accumulate the "points" for the negative features of the object version under evaluation; and (d) *Rewarding statement*: the *Action* is [Rewards *y*]. Such a statement expresses an accumulating rule in the policy specification to accumulate the "points" for the positive features of the object version.

(2) *(LoopControl:)?Fc(*C_1*,* C_2*, ...,* C_n*)*, the right hand side of a primary policy statement, consists of two parts, each of which is explained below.

(2.1)The first part, *(LoopControl:)?* is an optional loop control structure as indicated by the question mark (?). It controls recursive executions of the second part. More specifically, a loop control structure is in one of the following formats:

forAll variable; ST, Function(variable, other arguments)

forSome variable; ST, Function(variable, other arguments)

The terms, *forAll*, *forSome*, and *ST* (shorthand for *Such That*) are key words in *Selector*. *Variable* is called a control variable. *Function(variable, other arguments)* is called a control function and it specifies the allowed possible values for the control variable.

The semantic meaning of control structure is determined by the *forAll* and *forSome* loop control qualifiers. As implied by its name, *forAll* requires that every value of *variable* as specified by *Function(variable, other arguments)* satisfy *Fc(*C_1*,* C_2*, ...,* C_n*)*, i.e., *Fc(*C_1*,* C_2*, ...,* C_n*)* is evaluated as true for this *variable* value, in order to evaluate the right hand of the primary policy statement as true. *forSome* requires only one value of *variable* as specified by

Function(variable) make *Fc(C_1, C_2, ..., C_n)* true in order to evaluate the right hand side of the primary policy statement as true.

(2.2) The second part of the right hand of a policy statement, *Fc(C_1, C_2, ..., C_n)*, is an Boolean expression. Each argument, C_i, called a *conditional statement*, can be evaluated as either true or false. Conditional statements are combined by logical operators such as AND, and OR with the following format:

*Function(Arguments) (AND || OR Function(Arguments))**

The symbol "*" specifies that the term preceding "*" can appear multiple times.

Figure 4 shows the policy language statements (an instance of *Selector*) corresponding to the policy specification given in Figure 2.

(Primary statements)

1. Deny ← AffectedByVirus($V^{(O)}_i$)
2. Select ← (forAll *Variable*; ST, Component($V^{(O)}_i$, *Variable*)):
   Member(*Variable*, RecommendedObjectVersionList) AND
   FunctionCorrect(ComposingFunctions($V^{(O)}_i$));
3. Select ← Greater(T(S, $V^{(O)}_i$)$_{\text{Overall}}$, 0.95);
4. [Rewards 2] ← Member(Owner($V^{(O)}_i$), GoodContactList)
5. [Warnings 2] ← (forSome *Variable;* ST, Component($V^{(O)}_i$, *Variable*)):
   Less(Reputation(Owner(*Variable*)), 0.4)
6. [Rewards 2] ← (forSome *Variable*; ST, Component($V^{(O)}_i$, *Variable*)):
   Partner(Owner(*Variable*), S)
7. [Rewards 3] ← MemoryLeak($V^{(O)}_i$)

(Implied statements)

Deny ← Greater(*Number of Warnings*, 4)

Select ← Greater(*Number of Rewards*, 4)

Figure 4. The Set of Policy Statements Expressing the Policy Specification Rules Given in Figure 2. *AffectedByVirus, Component, Member, FunctionCorrect, ComposingFunctions, FunctionCorrect, Greater, Less, Partner, Owner,and MemoryLeak* are names of external library functions.

4.2 Implied Statements

Two implied statements are defined below

Select ← GreaterThan (Number of Rewards, Threshold $_1$)

Deny ← GreaterThan (Number of Warnings, Threshold $_2$)

Threadhold$_1$ and *Threshold$_2$* are supplied by the system administrators. The implied statements are applied to a given object version after all the primary statements have been evaluated towards the object version. If the implied statements still don't make a decision regarding that object version, it is added to a set, *Candidate*, which keeps the residual object versions for the target object.

4.3 The Residual Statements

Applying the primary policy statements to all the versions of object O leads to two possible results:

(1) A "*select*" decision has been made and the trust decision for O is completed; or

(2) No decision has been made and some versions have been added to *Candidate*.

The *residual statements* select a version from *Candidate*. One way is to select the version with the highest trustworthiness. In this case, the residual statement has the format:

$$V^{(O)} \leftarrow select\ V^{(O)}_i\ with\ Max(T(S,\ V^{(O)}_i)_{overall})$$

where $V^{(O)}$ represents the version selected by *S* for O, *select* and *with* are two key words, *Max* represents a function to select the maximum value of an input set, $V^{(O)}_i$ is an element of *Candidate*, and $T(S,\ V^{(O)}_i)_{overall}$ represents the overall trust value of $V^{(O)}_i$ for S, which can be calculated as the weighted average of the primary trust and secondary trust values of $V^{(O)}_i$ as below:

$$T(S,\ V^{(O)}_i)_{overall} = \lambda * T(S,\ V^{(O)}_i)_{primary} + \gamma * Trust(S,\ V^{(O)}_i)_{secondary}$$

where λ and γ are weights assigned to the primary and secondary trust values of $V^{(O)}_i$ for *S*. Those two trust values are explained next.

Definition 5: The *primary trust value* of an object version $V^{(O)}_i$ for an evaluator, such as a subject S, denoted as $T(S,\ V^{(O)}_i)_{primary}$, is the trustworthiness of $V^{(O)}_i$ for S in term of its quality and/or security, which is calculated based on S's direct experiences of studying the closely related information about $V^{(O)}_i$, e.g., the trustworthiness of the components of $V^{(O)}_i$ and the appropriation of composing functions used to form $V^{(O)}_i$.

Definition 6: The *secondary trust value* of $V^{(O)}_i$ for S, denoted as $T(S,V^{(O)}_i)_{secondary}$, is the trustworthiness of $V^{(O)}_i$ obtained through secondary experiences of S, e.g., the information S has on the trustworthiness of $V^{(O)}_i$ from other parties such as the owner of $V^{(O)}_i$, a recommender, or user evaluations.

$Trust(S,V^{(O)}_i)_{secondary}$ can be calculated, in its simplest form, as the mathematical product of the trust level of the object version for its owner, S' and the trust level of S' for S. The trust level of S' for *S* is called *subject trust* since it measures the trustworthiness of one subject for another. Several trust models have been proposed to evaluate the subject trust values (see [3] and [4] for more information). Other methods exist to calculate secondary trust value of an object based on transitive and discounted recommendations from third parties (see [2]). We will not discuss them here due to space constraints.

In order to calculate Trust(S, $V^{(O)}{}_{i})_{\text{primary}}$, S studies the component information of $V^{(O)}{}_{i}$, i.e., how it has been integrated, to what degree those components should be trusted, which set of composing functions have been used to form $V^{(O)}{}_{i}$, etc. Let elements in the set $\{C_1, C_2, ..., C_n\}$ denote the components of $V^{(O)}{}_{i}$ and elements in the set $\{S_1, S_2, ..., S_n\}$ represent the owners of those components with S_i being the owner of C_i for $0 < i \leq n$ respectively. T(S, $V^{(O)}{}_{i})_{\text{primary}}$ can be calculated by the following formula (see [1] for more information):

$$T(S, V^{(O)}{}_{i})_{primary} = \Gamma_{(S, V(O)i)}(F, T(S, C_1), T(S, C_2), ..., T(S, C_n))$$

where $\Gamma_{(S, V(O)i)}$ represents the trust function to evaluate Trust(S,$V^{(O)}{}_{i})_{\text{primary}}$ based on the trust values of the components of $V^{(O)}{}_{i}$, and $T(S, C_i)$ represents the trust value of component C_i for S, where $1 \leq i \leq n$; Trust function $\Gamma_{(S, V(O)i}$ takes composing functions represented by F and a set of component trust values as input. A trust function can provide answer to the question "how much should a compound object version be trusted given the trust values of its components and the composing functions used to form that compound object version?" Developing a general format for a trust function is both domain and user dependent. Two common cases are discussed below.

Case 1: For a weighted average composing function $F = (w_1 * C_1) + (w_2 * C_2) + ... + (w_n * C_n)$, the corresponding trust function is

$$T(S, V^{(O)}{}_{i})_{primary} = \Gamma_{(S, V(O)i)}(F, T(S, C_1), T(S, C_2), ..., T(S, C_n)) = w_1 * T(S,C_1) + w_2 * T(S,C_2) + ... + w_n * T(S, C_n)$$

where $w_1, w_2, ..., w_n$ are real numbers in the range [0, 1] and they add up to 1. Intuitively, if a composing function has the format as a weighted average, then the same parameters in the composing function are used to integrate the trust value of each component in order to calculate the trust value of the compound object version.

Case 2: For the composing function $F = c * A$, where c is a constant parameter, the corresponding trust function is

$$\Gamma_{(S, V(O)i)}(F, c, T(S, V^{(O)}{}_{i})) = T(S, V^{(O)}{}_{i})$$

5. OBJECT VERSION ATTRIBUTE VALUE DISCOVERY

The attribute values of a given object version are important for an evaluator to assess the trustworthiness of the object version in term of its quality and security. Attribute value discovery is to collect (calculate, test, analyze, or verify) the values of a set of attributes given an object version. Some trust attribute values of an object version can be calculated such as a owner's reputation and trustworthiness. Others can be dynamically tested, statically analyzed, or verified from the owner or a trusted authority.

The quality and security features of an object version can be tested by internal experts or external service providers. In either case, testing can be conducted statically or dynamically (see [21, 22, 23, 24, and 25] for more information about software feature testing). The former requires systematic analysis of the object version in term of its structures, algorithms, functions, etc. The source code of the object version is required for static analysis. Traditional methods for detecting security flaws include penetration analysis and formal verification of security kernels. Other general testing techniques include path testing, data-flow testing, and syntax testing. Dynamic analysis is to test the security and/or quality features of the object version in a controlled environment through a series of well-planned experiments.

Different from the above scenario, where the evaluator is responsible for verifying the quality and security features of a given object version, the ConCert project [19] requires that the producer of a code supply proofs for the security of the code. The project uses certificates to verify the security features of external programs and files. Certifiable policies cover type and memory safety (including system call or device access), control-flow safety, resource usage (CPU, Memory), abstraction boundaries, privacy and information-flow properties, and much more. Certifications are based on intrinsic properties of code, not the code producer's reputation. The proofs provided by those certificates are written in a specific machine-checkable form. There are several certified code systems: (1) Proof carrying code. Compiler produces a safety proof in logic and certification consists of proof checking; (2) Typed Assembly Language. Compiler produces type annotations for the machine code that imply safety and verification is type-checking. Both techniques work with native code and no expensive/complicated JIT compilation step is required. According to [88], the code developers follow the following procedure to supply certificated code: (1) start with program in safe language such as Java, SML, Safe C, (2) transform the code and simultaneously the reason that it is safe, and (3) finish with machine code, checkable certificate.

There are other forms of information certifications. For instance, information users can utilize redundancy check (or reworking) to verify the correctness of a given piece of information. For an active code, the provider can run the code on a given set of inputs and then the execution trace is used as a proof of the correctness of the code. For other types of information, native certifications may be used: theorem proof and facts or experiments. Those methods can be used in the proposed decision model.

6. CONCLUSIONS

This paper introduces a policy-oriented trust-based decision model for subjects to select reliable and secure information in an open system. It is a crucial step for achieving security and quality of service in such an open system where there is no single authority and where traditional security models do not work effectively. The proposed model allows a user to specify what features of external information it can't accept and what features are favorable to it. Based on this model, an example of a policy specification has been defined. *Selector*, a high-level policy language, has been developed to express the user-defined policy specification that allows automatic evaluation of the trustworthiness of available object versions of a given object and select one that meets the user's requirements for information quality and security. The paper also introduces object trustworthy calculations, which are important for users to make trust decisions. Compared with other decision-making approaches, our trust selection model is easy to understand and can be applied in computing systems. The model allows users to specify their customized policies to address their concerns for information integrity.

ACKNOWLEDGMENT

We are thankful to Dr. Robert L. Herklotz for his support, which made this work possible.

REFERENCES

1. Y. Zuo and B. Panda, "Component Based Trust Management in the Context of a Virtual Organization," In Proceedings of the 2005 ACM Symposium on Applied Computing, New Mexico, USA, March 2005

2. A. Josang, "An Algebra for Assessing Trust in Certification Chains," In Proceedings of the Internet Society 1999 Network and Distributed System Security Symposium, San Diego, USA, 1999

3. A. Rahaman, S. Hales, "Supporting Trust in Virtual Communities," In Proceedings of the 33rd Hawaii International Conference on System Sciences, Hawaii, USA, 2000

4. I. Ray, S. Chakraborty, "A Vector Model of Trust for Developing Trustworthiness Systems," In Proceedings of the 9th European Symposium on Research in Computer Security, Sophia Antipolis, French Riviera, France, 2004

5. T. Yu, X. Ma, M. Winslett, "PRUNES: An Efficient and Complete Strategy for Automated Trust Negotiation over the Internet," In Proceedings of the Conference on Computer and Communication Security, Athens, Greece, 2000

6. T. Yu, and M. Winslett, "Interoperable Strategies in Automated Trust Negotiation," In Proceedings of the Conference on Computer and Communication Security, Philadelphia, USA, 2001
7. W. Winsborough, N. Li, "Towards Practical Automated Trust Negotiation," In Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, IEEE Press, Monterey, USA, June 2002
8. L. Xiong, L. Liu, "A Reputation-based Trust Model for Peer-to-Peer E-Commerce Communities," In Proceedings of the IEEE Conference on E-Commerce, Newport Beach, California, USA, 2003
9. B .Yu, M. P. Singh, "Towards a Probabilistic Model of Distributed Reputation Management," In Proceedings of the 4th Workshop on Deception, Fraud and Trust in Agent Societies, Montreal, Canada, 2001
10. L. Mui, M. Mohtashemi, A. Halberstadt, "A Computational Model for Trust and Reputation," In Proceedings of the 35th Hawaii International Conference on System Science, Hawaii, USA, 2002
11. "An Introduction to Cryptography, in PGP 6.5.1 User's Guide," Network Associates Inc., p.11-36, http://fi.pgpi.org/doc/pgpintro/
12. Adams, C. and S. Farrell, "RFC2510 – Internet X.509 Public Key Infrastructure Certificate Management Protocols" http://www.cis.ohio-state.edu/htbin/rfc/rfc2510.html, 1999
13. Feigenbaum, J., "Overview of the AT&T Labs Trust Management Project: Position Paper," In Proceedings of the 1998 Cambridge University Workshop on Trust and Delegation, UK, 1998
14. Blaz, M., "Using the KeyNote Trust Management System," AT&T Research Labs, http://www.crypto.com/trustmgt/kn.html, 1999
15. Chu, Y.-H., J. Feigenbaum, B. LaMacchia, P. Resnick and M. Strauss, "REFEREE: Trust Management for Web Applications," AT&T Research Labs, http://www.farcaster.com/papers/www6-referee, 1997
16. P. McDaniel, "On Context in Authorization Policy," In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies*, Como, Italy,June 2003

17. N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," In Proceedings of the Policy Workshop 2001, Bristol, UK, January 2001

18. L. Kagal, "Rei: A Policy Language for the Me-Centric Project," HP Labs Technology Report, 2002

19. Chang, B., Crary, K., DeLap, M., Harper, R. and Liszka, J., "Trustless Grid Computing in ConCert" http://www.cs.cmu/~concert/talks/Murphy2002Trustless/trustless.ppt#1
20. M. Bishop, "Computer Security – Art and Science," Addison-Wesley, 2003

21. C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A Taxonomy of Computer Program Security Flaws," *Computing Surveys*, 26(3): pp. 211-255, 1994

22.K. Ashcraft and D. Engler, "Using programmer-written Compiler Extension to Catch Security Holes," In *Proceedings of 2002 IEEE Symposium on Security and Privacy*, pp. 143-159, Berkeley, CA, USA, 2002

23.M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems*, 9(2), 1996

24.H. Chen, H. and D. Wagner, "An Infrastructure of Examining Security Properties of Software," In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, Washington DC, USA, 2002

25. B .V. Chess, "Improving Computer Security Using Extending Static Checking," In *Proceedings of 2002 IEEE Symposium on Security and Privacy*, pp. 160-173, Berkeley, CA, USA, 2002