

Formalising Key Distribution in the Presence of Trust using Object-Z

Benjamin W. Long

Software Verification Research Centre
The University of Queensland,
Brisbane, QLD 4072,
Email: benl@svrc.uq.edu.au

Abstract

Key distribution and trust management are two areas of interest in the world of internet security and are often merged to provide key distribution in the presence of trust relationships. We present two generic models using the Object-Z formalism, one for key distribution and another for trust management, and we show how the models can be combined via inheritance to produce concise models of key distribution in the presence of trust. We use Pretty Good Privacy as a specific example.

Keywords: Key Distribution – Trust Management – Formal Specification – Object-Z

1 Introduction

Key distribution (Diffie & Hellman 1976) and trust management (Grandison & Sloman 2000) are two areas of interest in the world of internet security. Key distribution is required to inform users (of public key cryptography) of other users' public keys, and trust is used in several applications where users' behaviour is determined by the amount of trust they place in others to perform specific tasks. When distributing public keys, each key is distributed with the corresponding identity of the user to whom the key belongs. For communication to be secure, users must know that each key-identity pair is valid, i.e., that the identity corresponds to the real owner of the key. Users of applications are not always known to each other. Therefore, they often need to trust other users to validate keys for them. To allow for this, trust management is required in key distribution applications.

It is important that key distribution applications are safe to use. We have witnessed several attacks on software that was once believed to be secure (Schneier 2000). Formal methods have often been promoted as a way of improving the correctness of software. Previous work on formalising key distribution involves modelling specific key distribution protocols and verifying that properties such as *confidentiality* and *authentication* are maintained (Fidge 2001, Meadows 1995). Previous work on formalising trust has involved the use of logics and logic-based techniques (Grandison & Sloman 2000) to define how agents should behave based on knowledge of *trust relationships* (how much entities trust each other). We are motivated to produce an abstract model of key distribution that also captures trust, allowing us to formally specify agents' decisions (such as deciding whether to trust a key-identity pair) based

on the level of trust they have in others (to validate key-identity pairs).

We present two generic models using Object-Z (Duke & Rose 2000): one for key distribution and another for trust management. We show how the generic trust model is easily extended using the object-oriented inheritance feature to provide appropriate functionality for specific applications. For example, we extend the model to allow derivation of trust from recommendations. We then show how the key distribution model and trust management model can be combined using inheritance to produce concise models of key distribution in the presence of trust. We use Pretty Good Privacy (Network Associates, Inc. 1999) as a specific example as it is a relevant application.

2 Related Work

Several formal models of *key distribution protocols* have been presented (Fidge 2001, Meadows 1995). These involve the use of logics (including the BAN logic), CSP, the spi calculus, and the B and Z specification languages. Using such formal methods gives us the ability to model protocols and then conduct rigorous proofs to verify their security. However, these already complex models do not incorporate the concept of trust.

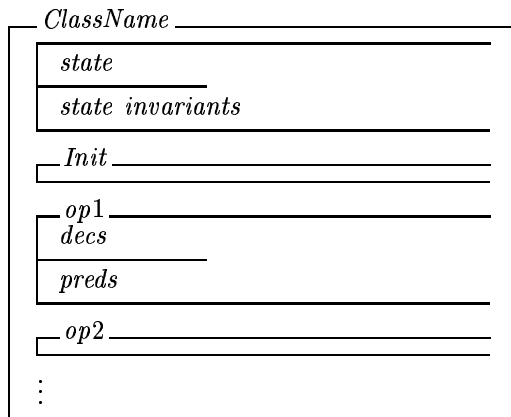
Formal models of trust have been specified using standard logics and other logic-based formalisms (Grandison & Sloman 2000). They often support knowledge such as '*a* believes *b*' and '*a* trusts *b*'. Such models (Weeks 2001, Grandison & Sloman 2001, Li, Feigenbaum & Grosz 1999, Jajodia, Samarati & Subrahmanian 1997, Jones & Firozabadi 2001) are concerned mainly with trust-based authorisations. However, again these models are already very complex and they seldom integrate specific applications such as key distribution. Beth *et al.* (Beth, Borcherting & Klein 1994) devised possible ways of deriving transitive trust relationships by applying intricate calculations to existing direct trust relationships.

Henderson *et al.* (Henderson, Coulter, Dawson & Okamoto 2002) model trust structures for public key infrastructures using set theory and logic, an approach similar to ours. However, their notion of trust is based on restriction of the *certification path*. In other words, an agent trusts a key only if it has been delivered via a number of agents less than the chosen threshold. This seems to be one of the few formal models that captures both trust and key distribution in the one model. Although restriction of the certification path is often a requirement in public key infrastructures, there is no ability to specify the amount of trust that one agent of the system has in another as described in other trust management models. Our goal is to devise a formal model that supports this.

3 Background: Object-Z

Object-Z (Duke & Rose 2000) is a mathematical language built on the state-based Z specification language. It provides constructs suitable for specifying systems in an object-oriented style. It can be used to describe the exact behaviour of systems without having to describe every detail such as implementable data structures and algorithms. Z's mathematical base helps prevent ambiguity and allows us to conduct rigorous arguments to establish desired properties of the specification (Potter, Sinclair & Till 1991).

The main construct used in Object-Z is the class schema which is used to specify a single type of object. In the class schema we specify the state of the class and any invariants on the state (using predicate logic) inside an unnamed schema box. We also specify the operations (*op1*, *op2*, etc.) that can be performed on the state, and an initialisation schema *Init* that specifies the state of the object when instantiated. The operation schemas each have a list *decs* of variable declarations, and a list *preds* of predicate conjuncts describing behaviours and constraints on the variables listed in the declarations. Any number of objects (instances of classes) can be declared within a 'system' class in order to specify how they interact with each other.



The advantage of using Object-Z for our purposes is that we can make use of object-orientation concepts such as abstract classes and inheritance in order to produce elegant specifications more applicable to object-oriented implementations such as a group of communicating agents.

4 Background: Public Key Cryptography

Public-key cryptography (Diffie & Hellman 1976) assumes all agents have a pair of corresponding keys associated with them: a public key that is distributed to everyone, and a private key that is only known to themselves. If a public key is used to encrypt a secret message, only the corresponding private key can be used to decrypt the message. Therefore, in order to send a secret message, the sender uses the recipient's public key to encrypt the message. Because the recipient is the only agent with the corresponding private key (the only agent that can decrypt the message), confidentiality is maintained.¹

Similarly, if a private key is used to encrypt a message, only the corresponding public key can be used to successfully decrypt it. An agent would encrypt a message with his private key in order to prove his identity to the recipient. A message that has been encrypted with a private key is commonly known as

¹We assume that cryptographic keys and algorithms are not compromised.

a digital signature. Like physical signatures, digital signatures are used to give evidence of the authority of a message. It is for this reason that digital signatures can be used for identification. If the recipient successfully decrypts the signed message using the corresponding public key, he knows the identity of the agent that encrypted the message because only that one agent should know the private key.

Digital signatures can also be used to ensure integrity of messages. If a message is signed and sent with the original text, the recipient can decrypt the signed message and compare it with the original text. If they are the same, the recipient knows that the message has not been modified in transit because it is unlikely that an intruder could modify both the original text and the signature consistently.

A further advantage of digital signatures is that they can authenticate the identity of the sender to anyone. Therefore, an agent can protect himself against the threat of dispute (Diffie & Hellman 1976). If a malicious agent tries to deny that he created and sent a particular message (*non-repudiation*), the signature provides evidence that the agent actually created it. Conversely, a malicious agent is unable to support a claim that another agent said something.

5 Modelling Key Distribution

In large networks, agents need a way of finding other agents' public keys. According to Kohnfelder (Ellison 1999), "Public-key communication works best when the encryption functions can reliably be shared among the communicants (by direct contact if possible). Yet when such a reliable exchange of functions is impossible the next best thing is to trust a third party."

Diffie and Hellman (Diffie & Hellman 1976) suggested that public keys should be stored in a public file along with the name and address of the agent to whom the key belongs. The concept of a public file has grown into what we now call a trusted third party (TTP) or certification authority (CA). An agent can find the public key for a given agent by sending an appropriate request to the CA. The CA responds by sending a message containing the public key and identity of the requested agent. The message is signed to indicate that the pair has been validated by the CA, and also to ensure integrity of the information because it is important that agents associate keys and identities correctly. These signatures created by the CA are referred to as certificates and were first mentioned by Kohnfelder (Ellison 1999). Encryption of the message is not required because public knowledge of one's public key is not harmful. There is often additional information in the certificate such as an expiry date or lifetime for the key (Henderson et al. 2002).

The CA's signature still does not ensure that the information contained inside the certificate can be trusted — a trusted party is one that handles its own private keys well (Ellison & Schneier 2000). Only if the CA is *trustworthy*, can agents confidently associate keys with identities. A CA does not need to be a well known authority designated solely for the purpose of validating keys. CAs can be other friendly agents believed to be trustworthy.

For CAs to be trustworthy, they must correctly validate key-identity associations inside the certificates that they sign and they must not have malicious intentions. Correctly validating a key-identity pair is one of the most critical procedures in key distribution. Depending on the situation, the validation process may be different. Authoritative CAs usually require a formal application process (Ellison & Schneier 2000, Gerck & MCG 1998) whereby applicants present some form of ID together with their

public keys. Non-authoritative CAs might not require such a formal process. In the case where the two parties are good friends, no process may be required. In any case, trust plays a key role in the validation process.

A system that supports the distribution, management and use of keys to provide confidentiality, authentication and other security properties is commonly referred to as a public-key infrastructure (PKI) (Older & Chin 2002). Applications such as X.509 (ITU-T Recommendation X.509, ISO/IEC 9594-8 1997), PGP (Network Associates, Inc. 1999), SPKI (Ellison, Frantz, Lampson, Rivest, Thomas & Ylonen 1999) and SDSI (Rivest & Lampson 1996) have been proposed for a PKI to promote and enhance the use of public-key cryptography in various fields of industry.

5.1 An Object-Z Model of Key Distribution

Each agent involved in key distribution (CA or user) has similar key distributing capabilities. Therefore we specify a class of agent *KeyDistributor* that has key distributing capabilities. We begin by introducing the set *Key* of all public keys in the system as a given type,

[*Key*]

and the set *Agent* of agents in the system. *Agent* is made up from the different types of agents that we specify in the sections to follow.

$$Agent == KeyDistributor \cup Suspicious \\ \cup Skeptical \cup Naive \cup PGP$$

A *KeyDistributor*, as defined below, has a *keyring* that stores all of the keys that the *KeyDistributor* knows about. (When using Object-Z, the information captured in the state schema is knowledge local to each instance of the class.) The *keyring* relation associates a key-identity pair with those agents who have validated the pair. Being a relation, the *keyring* does not necessarily have a set of validators for every possible key-identity pair. This allows for distinction between a key-identity pair with no validators, and a pair that is not yet known by the agent. Initially, the *KeyDistributor* does not know of any keys. This initialisation is captured by the *Init* schema.

By validating a key-identity pair, an agent states that they believe the key really belongs to the corresponding identity. We do not state how the key-identity pairs are validated. The validation process may be modelled for a specific application in a subclass of this generic key distributing agent. Generally, an agent indicates that he has validated a key by signing the key-identity pair in order to create a certificate. We have simplified the model by assuming that a key and identity are the only two items in a certificate, however, certificates may contain more data than this (Henderson et al. 2002).

A key distributing agent may add a key and remove a key (and associated information) to and from its keyring. An agent may also get a new key for itself. In order to tell other agents about keys on its keyring, a key distributing agent may give a known key to an agent.

<i>KeyDistributor</i>
(Init, addKey, removeKey, giveKey, newKey)
<i>keyring</i> : (<i>Key</i> × <i>Agent</i>) ↔ <i>Agent</i>

<i>Init</i>
<i>keyring</i> = ∅
<i>addKey</i>
Δ(<i>keyring</i>)
<i>key?</i> : <i>Key</i> ; <i>agent?</i> : <i>Agent</i>
<i>validators?</i> : ℙ <i>Agent</i>
<i>keyring</i> ' = <i>keyring</i> ∪ { <i>v</i> : <i>validators?</i> • (<i>key?</i> , <i>agent?</i>) ↦ <i>v</i> }
<i>removeKey</i>
Δ(<i>keyring</i>)
<i>key?</i> : <i>Key</i> ; <i>agent?</i> : <i>Agent</i>
<i>keyring</i> ' = {(<i>key?</i> , <i>agent?</i>)} ≀ <i>keyring</i>
<i>createKey</i>
<i>key!</i> : <i>Key</i> ; <i>agent!</i> : <i>Agent</i>
<i>validators!</i> : ℙ <i>Agent</i>
<i>agent!</i> = self
<i>validators!</i> = {self}
<i>giveKey</i>
<i>key!</i> : <i>Key</i> ; <i>agent!</i> : <i>Agent</i>
<i>validators!</i> : ℙ <i>Agent</i>
<i>validators!</i> = <i>keyring</i> ({(key!, agent!)})
<i>newKey</i> ≐ <i>createKey</i> <i>addKey</i>

The *addKey* operation takes a *key?*, *agent?*, and a set of validating agents *validators?* as inputs (denoted by decorating the variables with '?'). The key-identity pair is added to the keyring (if not already on the keyring), and all *validators?* that are not already known as validators of the given key and identity are added appropriately. The inclusion of the declaration 'Δ(*keyring*)' indicates that the state of the *keyring* may change. The post-state variable *keyring'* represents the value of the *keyring* function after the operation. The set comprehension on the right defines the set of mappings from the given key and agent to each of the validating agents.

The *removeKey* operation removes all knowledge about a given key-identity pair. This is done using Z's domain subtraction operator '≀' which removes all mappings from *keyring* that have the given *key?* and *agent?* in the domain. This operation would be invoked when a key-identity pair is found to be incorrectly validated, or simply when a key has expired and is no longer valid for the corresponding agent.

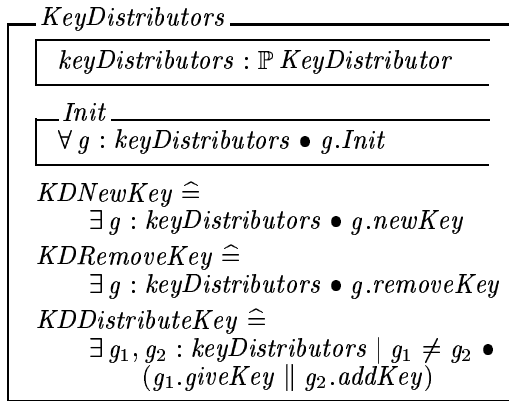
The *createKey* operation specifies the output (denoted by decorating the variables with '!') of a *key!*, an identity *agent!*, and set of *validators!*. As the key is a new key for the agent invoking the operation, the *agent!* to whom the key belongs is *self*. (The identifier *self* is an Object-Z identifier used by a class to refer to itself.) The agent is the only validator (*validators!* = {*self*}) for its new key. The value of the key is unspecified as it may be any key from the set *Key* of all keys. Note that there is no restriction on how many times the same key can be generated. In practice, it is assumed that a 'unique' key is generated by selecting a key at random from a large set of keys.

The *giveKey* operation specifies the output of a known key *key!*, with the corresponding owner's identity *agent!*, and the set *validators!* of agents that have validated the key-identity pair. The predicate *validators!* = *keyring*({(key!, agent!)}) ensures that the key, agent, and corresponding set of validators are on the agent's *keyring*. Z's operator (| ... |) denotes

the relational image of the *keyring* function.

Sometimes it may be undesirable for some operations to be invoked by other objects. For example, when specifying a system of key distributing agents, the *createKey* operation must be invoked in conjunction with *addKey* for the operation to be effective. A visibility list $\uparrow(\dots)$ identifies the operations available for use within other classes (excluding subclasses which inherit all operations). Therefore, we leave the *createKey* operation out of the visibility list and instead provide the ‘visible’ operation *newKey* which combines the two composite operations, *createKey* and *addKey*, using Object-Z’s parallel composition operator ‘||’ (Duke & Rose 2000). This operator equates the input variables from one schema with the output variables of the other, where the variable names (excluding the decorations) are the same.

The *KeyDistributors* class below, then specifies how several key distributing agents cooperate with each other.



The class has a set *keyDistributors* of *KeyDistributor* objects. Initially all agents *g* from the set of *keyDistributors* are initialised (*g.Init*). The *KDNewKey* operation allows one agent *g* from the set of *keyDistributors* to create a new key for itself and store it on its *keyring*. The *KDRemoveKey* operation allows one agent *g* from the set of *keyDistributors* to remove a key from its *keyring*. Thirdly, the *KDDistributeKey* operation allows an agent *g*₁ to give a key to an agent *g*₂ which receives the same key. We assume the said key is agreed on by both parties but we don’t specify how. Note that absence of a visibility list means that all operations in the *KeyDistributors* class are visible.

6 Modelling Trust

Trust is a major factor in business transactions. A person must be able to trust others that they do business with in order to run a business effectively. Business entities are often suspicious of each other, and therefore assign a level of trust either physically or mentally to other entities. Using such levels, one can then base business decisions and transactions on the level of trust for the people involved. As online business transactions are becoming commonplace, online trust is becoming a serious issue in several internet applications.

One can establish a level of trust for an agent either directly or through a recommendation. Direct trust is usually based on relationships, past experiences, or simply one’s opinion. However, when it is impossible or inconvenient for agents to establish a direct trust level for particular agents, they may require recommendations from other trustworthy agents. The sequence of agents required to find a derived trust level for a given agent is called the *recommendation chain* (Lamsal 2001). The longer the

recommendation chain becomes, the lower the level of trust should be.

Grandison *et al.* (Grandison & Sloman 2000) provide a survey of trust in internet applications. They define trust as “the firm belief in the competence of an entity to act dependably, securely, and reliably within a specified context.” A principal (truster) may have a certain level of trust in any number of other entities (trustees) and that level of trust may be different for each trustee. According to Grandison *et al.*, there is a need for a formalised approach for specification and reasoning about trust, especially since the amount of direct human interaction in systems is decreasing.

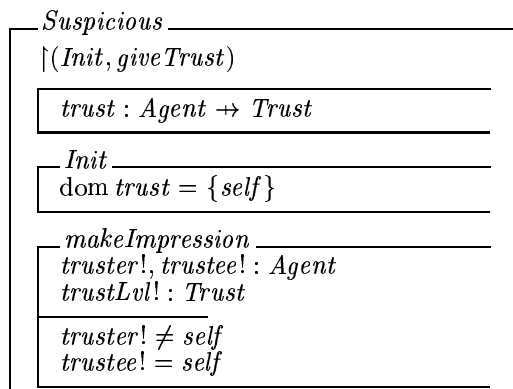
6.1 An Object-Z Model of Trust

Agents may assign levels of trust to other agents within a system. Some existing models use discrete values and some use continuous values to indicate the level of trust placed on an agent. Pretty Good Privacy (Network Associates, Inc. 1999), for example, uses four discrete values: implicit trust, complete trust, marginal trust, and no trust. To allow for the modelling of a variety of applications, we use a continuous value because it can be converted to a discrete value as required.

Beth *et al.* (Beth *et al.* 1994) use a real number between 0 and 1 (inclusive) for the level of trust, where 1 is the highest level (complete or blind trust) and 0 is the lowest level (no trust). Using such a measurement they are able to make complex calculations based on knowledge of several agents’ trust relationships. They state that trust levels should decrease as the recommendation path grows. Using such values for trust, they are able to achieve this inverse property by multiplying all trust levels in a recommendation path. For this reason, we use this range for specifying levels of trust in our model also. We define the set *Trust* of all real numbers \mathbb{R} between 0 and 1 (inclusive).

$$\textit{Trust} == \{v : \mathbb{R} \mid 0 \leq v \leq 1\}$$

We specify the generic suspicious agent class *Suspicious* that is concerned with trust below. (Remember that this is knowledge local to each individual agent.) The *trust* function stores a *Suspicious* agent’s trust in other agents. It is a partial function which means that not every agent has a level of trust associated with them. (An agent can not be trusted to any level until they are known.) We have simplified the model by assuming that agents are trusted to do all actions. To cater for a broader range of internet applications (Grandison & Sloman 2000), the agent could specify a level of trust for a given agent to do a given action. Initially, an agent trusts itself, as shown by the *Init* schema. Note that an agent does not necessarily trust itself completely — an agent may know that their validation process could be improved.



$\textit{recommend}$ $\textit{truster!}, \textit{trustee!} : \textit{Agent}$ $\textit{trustLvl!} : \textit{Trust}$
$\textit{truster!} = \textit{self}$ $(\textit{trustee!} \mapsto \textit{trustLvl!}) \in \textit{trust}$
$\textit{giveTrust} \hat{=} \textit{makeImpression}$ \vee $\textit{recommend}$

Direct trust is based on open world events such as relationships, past experiences, and opinions. The *makeImpression* operation is used to capture these concepts. The *truster!* is the agent obtaining the impression (*truster!* \neq *self*) and the *trustee!* is the agent giving the impression (*trustee!* = *self*). The variable *trustLvl!* represents the level of trust that the *truster!* has in the *trustee!* when obtaining an impression. The value of *trustLvl!* is not specified here because it is determined by any number of open world events.

Agents may recommend a trust relationship by revealing how much trust *trustLvl!* they (the *truster!*) place on another agent (the *trustee!*). The *recommend* operation specifies this behaviour. The predicate ensures that the *trustee!* and corresponding level are taken from the agent's *trust* function.

We group the two operations into one visible operation *giveTrust*. When giving information about trust, an agent is either making an impression in order to give a trust level for themselves or recommending someone else.

We have not specified how trust is received by an agent. This is because such functionality is application specific. However, given the 'abstract' *Suspicious* class, we can construct subclasses with extended functionality. For example, sometimes agents may be required to derive trust from direct relationships or indirectly through knowledge of other agents' trust relationships. One may call such an agent naive. However, such derivations of trust may be necessary in some situations and so should be considered in the analysis of trust systems (Grandison & Sloman 2000).

We specify the class *Naive* (Figure 1) with the ability to derive trust relationships from existing known relationships. A naive agent is a suspicious agent that stores knowledge of all known trust relationships in a function *trustRelationships*. (Inheritance is denoted by stating the name of the inherited class *Suspicious* at the top of the subclass.) From these relationships, derived relationships are evaluated and stored in a relation *transTrust*. It is a relation because it specifies every level of trust (direct and derived) for a given truster and trustee, therefore, possibly having multiple values for a given pair of agents.

The first three lines of the predicate in the state schema calculates the transitive values. For instance, if agent *A* directly trusts agent *B* to a level of 0.9 and agent *B* directly trusts agent *C* to a level of 0.7, then agent *A* trusts agent *C* to a level of $0.9 * 0.7 = 0.63$ transitively. If multiple values are derived for the one agent, they are all considered to be derived levels of trust for the agent and are therefore all present in the *transTrust* relation. This is one example of how to derive trust based on the work of Beth *et al.* (Beth *et al.* 1994) The last two lines in the predicate part of the state schema specify that, given several derived levels for an agent *g*, the maximum of the levels is used for the actual level of trust placed on the agent (*trust(g)*). This is not the only way to evaluate the derived levels — Beth *et al.* (Beth *et al.* 1994) use averages.

A *Naive* agent's *obtainTrust* operation allows the agent to get impressions of other agents, and also to

obtain knowledge about other agents' trust relationships indirectly. The received knowledge is added to the *trustRelationships* function using Object-Z's function override operator ' \oplus '. Inclusion of *transTrust* and *trust* in the Δ -list indicates that these functions may change. This is required for them to be updated in accordance with the state invariant.

An alternative to a *Naive* agent is a skeptical agent *Skeptical* which is a *Suspicious* agent that may obtain trust via first-hand impressions only.

$\textit{Skeptical}$ $\{(\textit{Init}, \textit{giveTrust}, \textit{obtainTrust})\}$ $\textit{Suspicious}$ $\textit{obtainTrust}$ $\Delta(\textit{trust})$ $\textit{truster?}, \textit{trustee?} : \textit{Agent}$ $\textit{trustLvl?} : \textit{Trust}$ $\textit{truster?} = \textit{self}$ $\textit{trust}' = \textit{trust} \oplus \{\textit{trustee?} \mapsto \textit{trustLvl?}\}$

Like the *KeyDistributor* class, we intend to use instances of these agents within a 'system' class to demonstrate how they are meant to interact with each other. By including *truster? = self* as a predicate in the *obtainTrust* operation, the agent *trustee?* giving trust will know to make an impression (to invoke the *makeImpression* operation and not the *recommend* operation).

7 Example: Pretty Good Privacy (PGP)

Pretty Good Privacy or PGP (Network Associates, Inc. 1999) is a PKI application that allows users to manage and distribute keys for secure communications. In the PGP world or *web of trust*, trust is managed not by certification authorities but by the users themselves. Each user chooses who they do and do not trust. In doing this, users effectively choose their 'CAs'. Trust is built from past and current relationships with people rather than from an "authority" that one does not even know. PGP uses certificates for secure distribution of keys. However, instead of being signed by the one CA, PGP certificates may contain multiple signatures created by various PGP users that have validated the key-identity pair. A PGP user will trust a key-identity pair if accompanied by the signature of a user that is trusted. Note that trust is not transitive in PGP. This means that if an agent *A* trusts agent *B* to validate keys, and *B* trusts agent *C* to validate keys, *A* does not necessarily trust agent *C* to validate keys.

Users store all known certificates that contain public keys on a *keyring*. Users may indicate how valid they believe each public key (key-identity) on their keyrings are and also the level of trust they place on the user of each key to validate other's keys. There are three levels of validity: *valid*, *marginal*, and *invalid*; and there are four levels of trust: *implicit*, *complete*, *marginal*, and *no trust*. Implicit trust is the trust one has in oneself. The other three levels are levels that can be assigned to other users' keys. The validity of a received certificate is based on the level of trust placed on the agents that have validated (signed) the key-identity pair. PGP requires one *completely* trusted signature or two *marginally* trusted signatures to establish a key as valid (Network Associates, Inc. 1999, pp. 32-33).

<i>Naive</i>
$\uparrow(\text{Init}, \text{giveTrust}, \text{obtainTrust})$
<i>Suspicious</i>
$\text{trustRelationships} : (\text{Agent} \times \text{Agent}) \rightarrow \text{Trust}$ $\text{transTrust} : (\text{Agent} \times \text{Agent}) \leftrightarrow \text{Trust}$
$\forall v_1, v_2, v_3 : \text{Trust}; g_1, g_2, g_3 : \text{Agent} \bullet$ $((g_1, g_3) \mapsto v_1 \in \text{transTrust} \Leftrightarrow ((g_1, g_3) \mapsto v_1 \in \text{trustRelationships} \vee$ $(\{(g_1, g_2) \mapsto v_2, (g_2, g_3) \mapsto v_3\} \subseteq \text{transTrust} \wedge v_1 = v_2 * v_3 \wedge g_1 \neq g_3)))$ $\forall g : \text{Agent} \bullet ((\text{self}, g) \in \text{dom transTrust} \Leftrightarrow g \in \text{dom trust})$ $\forall g : \text{dom trust} \bullet (\text{trust}(g) = \max \text{transTrust}(\{(\text{self}, g)\}))$
<i>obtainTrust</i>
$\Delta(\text{trustRelationships}, \text{transTrust}, \text{trust})$ $\text{truster?}, \text{trustee?} : \text{Agent}; \text{trustLvl?} : \text{Trust}$
$\text{trustRelationships}' = \text{trustRelationships} \oplus \{(\text{truster?}, \text{trustee?}) \mapsto \text{trustLvl?}\}$

Figure 1: Specification of a naive agent class.

7.1 An Object-Z Model of PGP

Before we begin to model a PGP agent, we firstly introduce the types *PGPTrust* and *PGPValidity* to correspond to the PGP model.

$$\text{PGPTrust} ::= \text{IMPLICIT} \mid \text{COMPLETE} \\ \mid \text{MARGINAL} \mid \text{NOTRUST}$$

$$\text{PGPValidity} ::= \text{VALID} \mid \text{MARGINAL} \\ \mid \text{INVALID}$$

Now we define the function *pgpLvl* that associates a given level from *Trust* with the corresponding PGP trust level from *PGPTrust*. We have arbitrarily chosen 0.6 to be the threshold between complete trust and marginal trust. If this function was defined as a function local to the agent instead of a global function as we have done here, each agent could even set their own individual thresholds. If this were the case, the threshold at which each agent identifies a key as valid would be considered for the impression they make on others.

$\text{pgpLvl} : \text{Trust} \rightarrow \text{PGPTrust}$
$\forall v : \text{Trust} \bullet$ $\text{pgpLvl}(1) = \text{IMPLICIT}$ $0.6 \leq v < 1 \Rightarrow \text{pgpLvl}(v) = \text{COMPLETE}$ $0 < v < 0.6 \Rightarrow \text{pgpLvl}(v) = \text{MARGINAL}$ $\text{pgpLvl}(0) = \text{NOTRUST}$

A PGP agent *PGP* (Figure 2) is a *KeyDistributor* that does not allow transitive trust. Therefore a PGP agent is a *Skeptical KeyDistributor* agent. This is identified by inheriting the appropriate classes. A PGP agent has a set of key-identity pairs that it has validated. The function *validated* associates every validated key-identity pair to a validity level. The predicates in the state schema ensure that a PGP agent implicitly trusts itself (and does not have implicit trust in anyone else), and that an agent can be a validator for a given key on the *keyring* only if the agent believes that the key is *VALID*. The *Init* schema specifies that initially no keys have been validated.

A PGP agent requires either one completely trusted agent or two marginally trusted agents to have signed a key for it to be considered valid. Although

not often stated explicitly, we assume that existence of only one marginally trusted validator for a key will establish a key as marginally valid, and if all validators are not trusted, the key will be considered invalid. Given a *key?* and *agent?*, the *validateKey* operation accommodates these four cases:

1. if a validator *v* for the given key-identity pair is completely trusted, the key is given a level of *VALID*, the *validated* function is updated to include this new knowledge, and the agent adds itself to the set of validators for the pair on the *keyring*;
2. if two validators *v*₁ and *v*₂ for the given key-identity pair are marginally trusted, the key is again *VALID* and the state is updated appropriately;
3. if only one marginally trusted agent has validated the key, then the key is validated to the *MARGINAL* level and the *keyring* function remains unchanged;
4. if no trusted agents have validated the key, the key is *INVALID* and again the *keyring* is unchanged.

The *validateKey* operation should be invoked when a key is added to a PGP agent's *keyring* in order to assign an appropriate validity level, and when a key is removed in order to set the key's validity level to *INVALID*. When inheriting the *KeyDistributor* class it is possible to change the name of selected operations to redefine them for a more specific application. In the *PGP* class, *newKey* is renamed *oldNewKey* (*oldNewKey/newKey*), and the same is done for *addKey* and *removeKey*. Then the operations are defined to be the old operations followed by the *validateKey* operation. The operations are composed using *Z*'s sequential schema composition operator '∘' (Duke & Rose 2000). Note that the new *removeKey* operation will assign the given key a level of *INVALID* as there are no validators for an unknown (removed) key. With the old operations redefined in this way, there is no need for *validateKey* to be a visible operation.

Given the specification of a single *PGP* agent class, we can specify a system *PGPSystem* of PGP agents that demonstrates how they are supposed to interact with each other.

$\uparrow(\text{Init}, \text{giveTrust}, \text{obtainTrust}, \text{giveKey}, \text{addKey}, \text{removeKey}, \text{newKey})$
<i>Skeptical</i> $\text{KeyDistributor}[\text{oldNewKey}/\text{newKey}, \text{oldAddKey}/\text{addKey}, \text{oldRemoveKey}/\text{removeKey}]$
$\text{validated} : (\text{Key} \times \text{Agent}) \rightarrow \text{PGPValidity}$
$\forall g : \text{dom trust} \bullet g = \text{self} \Leftrightarrow \text{pgpLvl}(\text{trust}(g)) = \text{IMPLICIT}$ $\forall (k, g) : \text{dom keyring} \bullet (k, g) \mapsto \text{self} \in \text{keyring} \Rightarrow (k, g) \mapsto \text{VALID} \in \text{validated}$
<i>Init</i> $\text{validated} = \emptyset$
<i>validateKey</i> $\Delta(\text{validated}, \text{keyring})$ $k? : \text{Key}; g? : \text{Agent}$
$((\exists v : \text{keyring}(\{(k?, g?)\}) \mid v \in \text{dom trust} \bullet \text{pgpLvl}(\text{trust}(v)) = \text{COMPLETE}) \Rightarrow$ $\text{validated}' = \text{validated} \oplus \{(k?, g?) \mapsto \text{VALID}\} \wedge \text{keyring}' = \text{keyring} \oplus \{(k, g) \mapsto \text{self}\})$ $((\exists v_1, v_2 : \text{keyring}(\{(k?, g?)\}) \mid v_1 \neq v_2 \wedge \{v_1, v_2\} \subseteq \text{dom trust} \bullet$ $(\text{pgpLvl}(\text{trust}(v_1)) = \text{MARGINAL} \wedge \text{pgpLvl}(\text{trust}(v_2)) = \text{MARGINAL})) \Rightarrow$ $\text{validated}' = \text{validated} \oplus \{(k?, g?) \mapsto \text{VALID}\} \wedge \text{keyring}' = \text{keyring} \oplus \{(k, g) \mapsto \text{self}\})$ $((\exists v : \text{keyring}(\{(k?, g?)\}) \mid v \in \text{dom trust} \bullet \text{pgpLvl}(\text{trust}(v)) = \text{MARGINAL} \wedge$ $(\forall \text{other} : \text{keyring}(\{(k?, g?)\}) \mid \text{other} \neq v \wedge \text{other} \in \text{dom trust} \bullet$ $\text{pgpLvl}(\text{trust}(\text{other})) = \text{NOTRUST})) \Rightarrow$ $\text{validated}' = \text{validated} \oplus \{(k?, g?) \mapsto \text{MARGINAL}\} \wedge \text{keyring}' = \text{keyring}$ $((\forall v : \text{keyring}(\{(k?, g?)\}) \mid v \in \text{dom trust} \bullet \text{pgpLvl}(\text{trust}(v)) = \text{NOTRUST}) \Rightarrow$ $\text{validated}' = \text{validated} \oplus \{(k?, g?) \mapsto \text{INVALID}\} \wedge \text{keyring}' = \text{keyring})$
$\text{newKey} \hat{=} \text{oldNewKey} \wp \text{validateKey}$ $\text{addKey} \hat{=} \text{oldAddKey} \wp \text{validateKey}$ $\text{removeKey} \hat{=} \text{oldRemoveKey} \wp \text{validateKey}$

Figure 2: A PGP agent class.

PGPSystem
$\text{agents} : \mathbb{P} \text{PGP}$
<i>Init</i> $\forall g : \text{agents} \bullet g.\text{Init}$
$\text{PGPNewKey} \hat{=}$ $\exists g : \text{agents} \bullet g.\text{newKey}$
$\text{PGPRemoveKey} \hat{=}$ $\exists g : \text{agents} \bullet g.\text{removeKey}$
$\text{PGPDistributeKey} \hat{=}$ $\exists g_1, g_2 : \text{agents} \mid g_1 \neq g_2 \bullet$ $(g_1.\text{giveKey} \parallel g_2.\text{addKey})$
$\text{PGPEstablishTrust} \hat{=}$ $\exists g_1, g_2 : \text{agents} \mid g_1 \neq g_2 \bullet$ $(g_1.\text{giveTrust} \parallel g_2.\text{obtainTrust})$

There are four system operations. The operations, PGPNewKey , PGPRemoveKey , and PGPDistributeKey , are defined in the same way as for the KeyDistributors system class. However, the operations behave differently because they have been redefined to include the validateKey operation. For example, when a PGP agent invokes newKey , a key is created, added to the keyring and it is assigned a validity level. The fourth operation is PGPEstablishTrust which specifies an agent giving trust to another agent.

8 Conclusion

We have successfully modelled a known key distribution application that incorporates trust management

using the Object-Z formal specification language. By using Object-Z, we were able to model two distinct concepts (key distribution and trust management) in isolation and then we were able to integrate them in order to produce a model of the required form for the specific application (Pretty Good Privacy).

The generic KeyDistributor class specifies a single key distributing agent. It is modelled at a very abstract level with no constructs for modelling certificates or encryption methods. Agents that have signed key-identity pairs to vouch for the pair's validity are merely represented in our model as validators for the given pair. The concept of certificates and encryption methods could be integrated into our model to allow for the analysis of confidentiality and authentication properties.

The Suspicious class allows for agents to keep a record of how much they trust others. Using the object-oriented inheritance technique, we were able to specify two subclasses of the Suspicious class: a Naive agent that derives trust levels from second-hand information, and a Skeptical agent that only considers trust levels that have been established directly. We emphasise that our calculations for derived trust are only examples to illustrate the ability to calculate and use derived levels of trust within our framework. Beth *et al.* (Beth *et al.* 1994) presented a method for deriving trust from recommendations which we believe can also be captured in our formalism.

These classes provide the basic needs for specifying key distribution and trust management. With the flexibility of inheritance, it is simple to inherit the functionality of our classes, and to specify stronger restraints and more specific behaviour as required. For example, we specified the PGP agent by inheriting both the KeyDistributor class and the Skeptical class

in order to capture appropriate functionality for key distribution based on trust. Given this class, we then modelled the PGP system *PGPSystem*, demonstrating how instances of *PGP* agents interact with each other.

By modelling each individual security concept (and its associated properties) in a separate class, we can provide elegant specifications of systems that incorporate several such concepts. In this paper, we combined the two concepts, key distribution and trust management, for PGP. We believe that this approach will provide more complete and coherent specifications for a variety of security applications.

Acknowledgments

I wish to thank Colin Fidge, Antonio Cerone, Luke Wildman, and the anonymous referees for reviewing this paper and providing helpful comments, and Graeme Smith for help with Object-Z technicalities. Presentation of this paper was assisted by Australian Research Council Linkage Grant LP0347620, *Formally-Based Security Evaluation Procedures*.

References

- Beth, T., Borchering, M. & Klein, B. (1994), Valuation of trust in open networks, in 'Proc. 3rd European Symposium on Research in Computer Security - ESORICS '94', pp. 3-18.
- Diffie, W. & Hellman, M. E. (1976), Multiuser cryptographic techniques, in 'Proceedings of AFIPS 1976 National Computer Conference', Montvale, New Jersey, pp. 109-112.
- Duke, R. & Rose, G. (2000), *Formal Object-Oriented Specification Using Object-Z*, Cornerstones of Computing, Macmillan Press Limited, UK.
- Ellison, C. (1999), 'The nature of a useable PKI', *Computer Networks* 31(8), 823-830.
- Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B. & Ylonen, T. (1999), 'RFC 2693: SPKI certificate theory'. <ftp://ftp.isi.edu/in-notes/rfc2693.txt>. Accessed November 2002.
- Ellison, C. & Schneier, B. (2000), 'Ten risks of PKI: What you're not being told about public key infrastructure', *Computer Security Journal* 16(1), 1-7.
- Fidge, C. J. (2001), A survey of verification techniques for security protocols, Technical Report 01-22, Software Verification Research Centre, The University of Queensland, Brisbane.
- Gerck, E. & MCG (1998), 'Overview of certification systems: X.509, CA, PGP and SKIP'. <http://www.mcg.org.br/>. Accessed November 2002.
- Grandison, T. & Sloman, M. (2000), 'A survey of trust in Internet applications', *IEEE Communications Surveys*.
- Grandison, T. & Sloman, M. (2001), 'Specifying and analysing trust for internet applications'. <http://www.doc.ic.ac.uk/~tgrand/TrustPaper.pdf>. Accessed November 2002.
- Henderson, M., Coulter, R., Dawson, E. & Okamoto, E. (2002), Modelling trust structures for public key infrastructures, in 'Information Security and Privacy - ACISP 2002', Vol. 2384 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 56-70.
- ITU-T Recommendation X.509, ISO/IEC 9594-8 (1997), 'Information Technology - Open Systems Interconnection - The Directory: Authentication Framework'.
- Jajodia, S., Samarati, P. & Subrahmanian, V. S. (1997), A logical language for expressing authorizations, in 'Proceedings of the IEEE Symposium on Security and Privacy', Oakland, CA, pp. 31-42.
- Jones, A. J. I. & Firozabadi, B. S. (2001), *On the characterisation of a trusting agent - aspects of a formal approach*, Kluwer Academic Publishers, pp. 157-168.
- Lamsal, P. (2001), 'Understanding trust and security'. <http://www.cs.Helsinki.FI/u/lamsal/papers/UnderstandingTrustAndSecurity.pdf>. Accessed November 2002.
- Li, N., Feigenbaum, J. & Grosz, B. N. (1999), A logic-based knowledge representation for authorization with delegation, in 'Proceedings of The 12th Computer Security Foundations Workshop', Mordano, Italy, pp. 162-174.
- Meadows, C. A. (1995), Formal verification of cryptographic protocols: A survey, in 'Advances in Cryptology - ASIACRYPT '94', Vol. 917 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 133-149.
- Network Associates, Inc. (1999), 'PGP 6.5.1 User's guide in english - An introduction to cryptography'. <http://www.pgpi.org/doc/guide/6.5/en/intro/>. Accessed November 2002.
- Older, S. & Chin, S.-K. (2002), 'Formal methods for assuring security of protocols', *The Computer Journal* 45(1), 46-54.
- Potter, B., Sinclair, J. & Till, D. (1991), *An Introduction to Formal Specification and Z*, Prentice Hall International Series In Computer Science, Prentice Hall, London.
- Rivest, R. L. & Lampson, B. (1996), 'SDSI - A simple distributed security infrastructure', Presented at CRYPTO'96. <http://theory.lcs.mit.edu/~rivest/sdsi10.ps>. Accessed November 2002.
- Schneier, B. (2000), *Secrets & Lies - Digital Security in a Networked World*, John Wiley & Sons, Inc., US.
- Weeks, S. (2001), Understanding trust management systems, in 'IEEE Symposium on Security and Privacy', Oakland, California, pp. 94-105.