SECURE Validation

C. Bryce[1], V. Cahill[2], N. Dimmock[3], K. Krukow[4],
J.-M. Seigneur[1,2] and W. Wagealla[5]

[1]University of Geneva, [2]Trinity College Dublin, [3]University of Cambridge,
[4]University of Aarhus and [5]University of Strathclyde

Abstract. This deliverable reports on the *validation* results obtained after an evaluation of the SECURE project's trust-based security model.
The first step of the evaluation is the definition of *validation criteria* for the SECURE model. Trust-based security frameworks are increasingly popular, yet few evaluations have been conducted. As a result, no guidelines or evaluation methodology have emerged that define the measure of security of such models. One of the contributions of this deliverable is to propose a methodology.
A key feature of the validation is that it focusses on the whole SECURE approach, rather than on individual elements of the framework or its implementation. The aim of evaluation is to measure how all components – working together – help to make a system secure. Individual components are evaluated in other project deliverables.
One of the main validation results is that the trust-based approach is complementary to traditional approaches. In the case of a SPAM filter for instance, the best results are obtained by a combination of both approaches. The advantage of the trust-based approach is that it allows each entity in a system to evolve its policy dynamically based on observations of the behaviour of others. This implies an absence of a global security policy, which in turn makes a successful attack on a community of entities more difficult.

1 Introduction

The first two years of the SECURE project saw the development and implementation of a security model for autonomous (a.k.a. *global computing*) systems. This model, which is termed "SECURE" in the remainder of this document, has more in common with operational security in the human world than with traditional computer security models.

SECURE belongs the family of *trust-based* models that have emerged in the computer security field for access control and security decision optimisation, e.g., [12, 5, 35, 36]. Their goal is to permit interacting principals to build trust in one another, in an analogous way to humans. This approach removes the need for pre-configured access control rules, which is advantageous for two reasons.

1. Modern infrastructures do not necessarily have an administration that is capable of making security decisions for all principals, either due to the decentralised nature of the network or to the exceedingly large number of principals. This text book definition applies to a wide range of systems and applications, e.g., people-area and

body-area networks [14], embedded device networks [10] and Internet peer-to-peer application frameworks [24].
2. The trust-based approach scales better since the principals being controlled need not be known in advance; rather, the onus is placed on each principal to become known and trusted through its actions in the system.

The SECURE model was implemented during the course of the project as a software kernel[1]. A SPAM filter application is one of the applications coded over the kernel. Whereas Bayesian SPAM filters classify a mail message as SPAM based on a statistical analysis of phrases in the message body, the SECURE SPAM filter can classify a message as SPAM based on a number of criteria, including trust in the message sender, the risk incurred by having a valid mail marked as SPAM, or confidence in the underlying system's ability to recognise the sender of the mail. A key observation of this work is that SECURE can be implemented in a real autonomous system, and that, when used for SPAM filtering, the model provides a complementary approach to Bayesian filtering.

The SPAM filter however is just one application of SECURE, which was designed for autonomous systems and applications in general. Fundamentally, we have to ask ourselves the following question: *Is* SECURE *a good approach to enforcing security in autonomous system applications?*. The goal of the SECURE project is to attempt to answer this question, and our answer is reported in this deliverable.

Trust-based security frameworks are complex and, as a result, many researchers only address a small number of the problem-space's facets. As a result, we know of few systems which have implemented **and** completely evaluated. In undertaking to address this gap, it is necessary to define exactly *how* to go about analysing a complete computational trust-system. Part of the challenge in evaluating a trust-based security model is that the computation of a trust value for a principal is only the first step. The goal of any security model is to take a decision, so the remaining steps include establishing confidence in the identity or credentials of the requesting principal(s), and depending on the framework, calculating the risk involved in permitting or refusing the requested action. Only then is the final security decision taken.

In undertaking such an evaluation, the first task is to clearly delimit the role of each component of the security framework. This is particularly important for the trust model component since "trust" in the human world is subject to various interpretations, and arguments on its semantics can have a confusing impact on the evaluation of computational trust systems. Once the precise role of each component is clarified, an analysis of the component is undertaken to determine how failures of, or attacks on, the component influence the outcome of the decision making process of the security system. These attacks are added to a *system attack profile* against which the whole security system is empirically evaluated. By evaluating all components together, one can also measure the effectiveness of computational trust against weaknesses in other components, e.g., the usefulness of recommendations when principal authentication is weak.

In addition to measures of system security that can be made of the model, much attention needs to be paid to the ability to implement the model in a real autonomous system, as without this ability, the model has no value. The challenge in SECURE is that

[1] Deliverable D5.2

no device is under the control of a trustworthy party, so we need to understand how manipulations of the model implementation on a device impacts on the system's behaviour. A further challenge is to ensure that a device cannot corrupt the model implementation that runs on another device, e.g., by sending it a virus or worm software.

The deliverable follows both of these axes – measures of security and facility of implementation. The document is organised as follows. Section 2 addresses the first point mentioned concerning evaluation – how the model as a whole can be evaluated. Section 3 discusses these criteria in the context of SECURE, and the criteria are then used for the evaluation of a SPAM filter that is built using SECURE in Section 4. Section 5 then considers a second evaluation criteria, that of being able to implement the model in a real system, and we present our results in relation to SECURE in Section 6 and in the appendices. Once again, the example used is the SPAM filter. Section 7 presents related work, and Section 8 summarises the main validation results.

2 Evaluation Criteria

The fundamental purpose of SECURE and other trust-based frameworks is to automate a decision process for security. The decision to take is whether a requesting principal should be allowed access to a resource. Another interpretation of the decision process is that it should select an interaction partner principal such that the risk of interacting with that principal is deemed to be acceptable to the decision-maker.

Numerous security frameworks have been designed over the years to implement this decision making process, e.g., [5, 18, 13]. The originality of trust-based frameworks is that they include a *computational treatment of trust* in the process. There are thus two basic questions that an evaluation methodology must uncover:

1. *The Question of Security* How good is the framework at automating the decision making process, and how likely is it to take the right decision?
2. *The Question of Trust* How does the computational treatment of trust improve, or impact upon, the security question?

It is important to distinguish these two questions. The Security Question is the fundamental one, and depends not only on the trust component, but also on the framework's risk assessment and principal recognition or authentication components.

A basic assumption of researchers working on trust-based models is that these models yield better decision making for security. As mentioned by Langheinrich [21], some researchers seem to believe that the closeness of these models to the operational human trust model is a gauge of the models' strength. It is a rather strange assumption, especially since human judgement has a negative, error-prone connotation. In any case, it illustrates how measures of success for these models have not yet been fully thought out.

This section is divided into three parts. Sections 2.1 and 2.2 look at the security and trust questions respectively. Section 2.3 describes potential attack scenarios and weaknesses against which a system must defend itself.

2.1 The Question of Security

There are four major elements to all security infrastructures, as suggested in Figure 1. One element is the principal authentication or recognition phase. The goal of authentication is to ascertain the identity of the requesting principal; recognition is used when the system does not guarantee a fixed identity for principals, and seeks to determine if a requesting principal has been previously encountered [30]. Authentication and recognition are done at runtime, once the requesting principal furnishes its identifying information.

The goal of the risk assessment element is to attribute a cost to permitting or refusing each request, and perhaps the likelihood of that cost occurring. The cost might be represented as the loss of information, privacy being compromised or it might even be a monetary value. Risk assessment might be informally or intuitively done, but it is reflected in the set of access rights that principals are accorded. Trust attribution to a principal $\mathcal{P}$ is defined as the level of risk that one is willing to take with $\mathcal{P}$. Finally, the decision phase of a security framework decides whether a request from $\mathcal{P}$ should be permitted based on the risk and trust calculations.

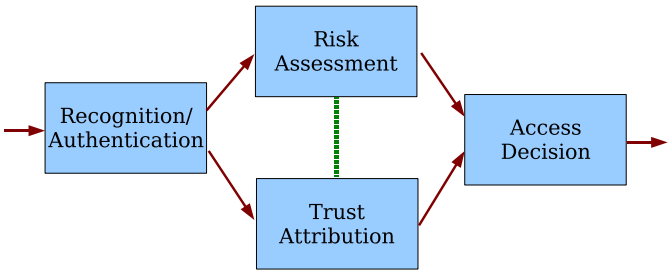

Fig. 1. Security Decision Framework

Context The key difference between traditional access control based security infrastructures, e.g., [18], and trust-based frameworks concerns when and how risk assessment and trust attribution are done.

In a Unix environment for instance trust attribution and risk assessment are implicitly made by a user when he decides what access rights to accord to users of his group and to others for his files. Generally, each user is accorded access to a file only if he really needs access to go about his work – this is also known as the *Principle of Least Privilege* [25]. The reasoning behind this principle is to restrict that damage that a principal can do, even if that principal is trustworthy but things go wrong. The trust value of each user is indirectly represented in these rights since they determine what actions the user is trusted to undertake. The decision phase in Unix simply checks whether the requesting principal, or his running OS process, has been accorded the necessary rights for the requested action.

In trust-based security frameworks, e.g., [23, 22], the trust decision is made at runtime, and is not guaranteed to be the same between two successive requests. It generally based on observations of the requesting principal's behaviour. Risk assessment is generally included in the decision policy design, where one defines the actions that are permitted for each of the possible trust values. This policy is defined as a mapping from trust values to permitted actions, and an implicit risk assessment guides this mapping. One of the originalities of the SECURE project is that the risk assessment phase is carried out at runtime.

Clearly, the difference between access control-based security frameworks and emerging trust models is that the former are *administration-oriented* while the latter are built for *autonomous* decision making. The former are preferred in environments where the users are known in advance and the owner of resources wishes complete control over the access that others have to his resources. The latter are required when there is no centralised control in the system, where the total number of principals cannot be identified or where there is no commonly trusted party to store security data. Here, the trust approach is required so that an unknown principal can show itself to be trustworthy and thus interact with others. Thus the *context* for the two classes of models is different. A context defines the set of use cases and *attack scenarios* against which a security model must be measured. An attack by a principal or group of principals is *any behaviour that aims to influence a security decision in such a way as to be detrimental to the decision maker*.

The John Smith Test The security subsystem implements a *decision-making process*. The obvious evaluation criteria is its ability to take the right decision. Consider the example of a Bayesian SPAM filter. Here, the strength of the mechanism is given by the number of correct message classifications, the number of false positives and false negatives. If we abstract away from the application-specific nature of this example, we can quantify the strength of a security mechanism by the triple $< \mathcal{E}, \mathcal{P}, \mathcal{N} >$. $\mathcal{E}$ represents the total number of events that need to be controlled by the security subsystem; $\mathcal{P}$ represents the number of false positives (decisions where access was incorrectly refused) and $\mathcal{N}$ represents the number of false negatives (decisions where access was incorrectly granted). The number of correct decisions is given by $\mathcal{E} - (\mathcal{P} + \mathcal{N})$. It is useful to distinguish $\mathcal{P}$ and $\mathcal{N}$ since the cost of a wrong decision can be different when a request is incorrectly refused from when it is incorrectly permitted.

The notion of *right decision* is not only application specific, but also varies from principal to principal. In the case of SPAM for instance, a message considered SPAM by one principal might be considered as valid by another. There is no universal notion of security in an autonomous system.

The notion of right decision is best determined by a, possibly imaginary, human user who surveys the system. This individual, whom we name John Smith, might be a system user or designer. He represents the person who just knows what the correct decision outcome is for a particular principal in a particular context. In this respect, the strength of the security subsystem is determined by a Turing-like test that is run for all events and which measures $\mathcal{P}$ and $\mathcal{N}$. The set of right and wrong replies for John Smith are contained within a data structure known as the John Smith *profile*.

Profile There are two profiles that are used when deciding the strength of a decision making process.

1. John Smith's profile, where John Smith is the decision making principal. His profile defines the correct outcome to each decision. That is, it defines the outcomes to the decision process that John Smith would give if he were asked personally rather than have the decision taken for him by the security framework.
2. The community profile is made up of the messages for John Smith from principals. These messages represent requests, recommendations and evidence. The community profile models all principals that can influence John Smith's security decision.

Different use case and attack scenarios against which the system is measured are modelled using different community profiles, e.g., a collusion attack can be modelled by a profile where different principals send incorrect recommendations for each other. $< \mathcal{E}, \mathcal{P}, \mathcal{N} >$ values are only comparable for the same profiles. We define $\mathcal{E}_{\mathcal{P}}$ as the profile, and refine our metric to $\mathcal{E}_{\mathcal{P}} \rightarrow < \mathcal{P}, \mathcal{N} >$, where the number of events of the profile $\mathcal{E}_{\mathcal{P}}$ is $\mathcal{E}$.

The Turing-test nature of the evaluation suggests that empirical measurements for a range of profiles are required to evaluate the strength of the security mechanism. Indeed, several systems have been evaluated in this manner, e.g., [12], and we use this approach ourselves in Section 4 to evaluate a SPAM-filter application that employs SECURE. There are however two important considerations:

1. The number of events $\mathcal{E}$ that all profiles generate can be very large or even unbounded, which makes a complete analysis unfeasible. When we use a subset of profiles, we must be able to argue that the subset used is sufficient to yield a meaningful measure of security. For this reason, the evaluation criteria are defined in terms of the specific attack scenarios that the model must withstand.
2. An analysis and measure of the value of security is still required, to be able to extract useful information from the myriad of simulation data. Simulation data alone do not tell us *why* results are good or bad. We need to understand how potential weaknesses in each of the components lead to incorrect decisions being taken. Other SECURE project deliverables elaborate on measures specifically related to components of SECURE.

Computational Cost Apart from the behaviour of the system in the presence of attacks, another useful measure is *cost*. The cost of the decision making process is measured with respect to CPU, memory and network, as well as more abstract resources such as the number of OS processes or databases employed. For general evaluation, there are three resources of particular importance. The first is *network*, which can be abstractly measured as the number of messages exchanged in the model, i.e., for evidence distribution. This is important to measure since security relies on being able to exchange information between principals – one thus needs to measure the level of security as a function of collaboration.

The second cost is *memory*, which can be abstractly measured as the number of items of evidence (observations and recommendations) in a principal's evidence store.

This aspect is important since autonomous computing devices might not have a lot of memory available. The importance of the figures for network and memory consumption is that they give a measure of the model's *scalability*. The final measure is CPU: though verifying the presence of an access right is single-instruction, policies can be expressed as code units, which means that the number of instructions becomes a measure. We aggregate these three costs to $\mathcal{C}$. We thus extend our security metric to $\mathcal{E}_\mathcal{P} \rightarrow < \mathcal{P}, \mathcal{N}, \mathcal{C} >$.

Implementation and Operation There is a difference between a framework *model* and its *implementation*. One could measure the ability of engineers to implement secure channels or a tamperproof kernel that can store data. This is a challenge in the design of any system. The challenge we undertook in SECURE was to implement the framework in a safe programming language so that the resulting kernel could be ported to any device. This issue is further considered in Sections 5 and 6.

Another measure is the ease with which one can express policies using the framework. The is important since most security breaches in system arise from incorrect use or non-use of the system's security mechanisms [1]. The limit of most users' policy expressions are the access control lists they set on their files in Unix. This nonetheless is deceptively easy: though bits on a file can be changed with the chmod command, there is no guarantee that the policy is meaningful from a security viewpoint.

Summary Comparison of trust based models to existing security architectures is not completely fair, since the latter are aimed at autonomous systems. To evaluate the model, one must identify the application specific profile $\mathcal{E}_\mathcal{P}$ that encapsulate the attacks that the framework is designed to withstand; the number of these events is $\mathcal{E}$, and the measure of security for these events is $\mathcal{E}_\mathcal{P} \rightarrow < \mathcal{P}, \mathcal{N}, \mathcal{C} >$. In a second stage, the role of each component needs to be clearly delimited so that the impact of each on the security decision process can be measured via associated attack scenarios.

2.2 The Question of Trust

Trust frameworks rely on evidence exchanged between principals. This allows a principal to learn about principals that are behaving badly. Traditional architectures do not have this possibility, so make no distinction between well-behaving and badly-behaving principals at runtime. *The added value of a trust model is to render it possible to make this distinction*.

No trust model can predict the future behaviour of a principal. The assumption behind the models where trust values encode history is that *the best available indicator of a principal's expected behaviour is its previous behaviour*; see Assertion T below. Evidence distribution systems seek to maximise the knowledge available to one principal concerning the behaviour of others. This is considered essential for security since, even if principal Alice believes Bob to be honest, she may learn that Bob was dishonest with Charlie, and thus revise her trust in Bob.

Assertion T The more one knows about a principal's previous behaviour, the better one can judge its future behaviour.

Assertion T encapsulates the fundamental strength and limitation of trust models. The same assertion is often employed in the human world. In any case, it shows that the trust model can do nothing against a random failure, i.e., where a well-behaving principal abruptly becomes malicious. The assertion sums up the best we can do concerning security, and represents a sort of *undecidability* for security based on trust based models. This also suggest that trust models alone are not the panacea for security, but are best combined with traditional approaches. This limit is illustrated by the line separating the *Security* state in Figure 2 from the others. The security state can be defined as $<$ **min**($\mathcal{P}$), **min**($\mathcal{N}$), **min**($\mathcal{C}$)$>$ for the set of profiles $\mathcal{E}_{\mathcal{P}}$ of the system under evaluation.

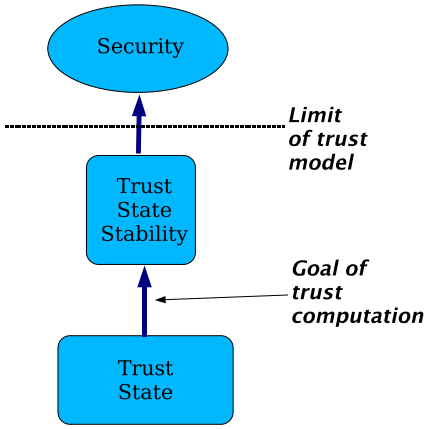

Fig. 2. Conceptual Limit of Trust-based Models

The intermediate state in Figure 2, denoted *trust state stability*, represents a state where every principal has an accurate account of the past behaviour of all other principals. The role of the trust model, and its evidence distribution framework, is to ensure a transition from any state to trust stability. *A measure of the trust component is its ability to achieve trust state stability or to reach a better understanding of principals' behaviour*. Many of the evaluations taken of trust models actually measure this, e.g., [23, 12, 22]; the measures given do not say anything about the overall security of the systems.

Trust stability in practice is very hard to achieve in a reasonably sized system for two reasons. First, there is simply too much information for principals to exchange and store. Second, principals are faulty and malicious and may be unable or unwilling to exchange their trust values. Further, principals can collude with false trust values, e.g., Alice creates false recommendations for Bob to help Bob cheat by making Charlie accord too much trust in him. A trust model must work with *partial and uncertain information*. Nevertheless, the exchange of trust values should bring the system closer

to stability. For this reason, the attack scenarios of $\mathcal{E}_{\mathcal{P}}$ must model insufficient numbers of, and false, recommendations.

2.3 Validation Criteria

As we have just seen, the measure of any security model is the ability of any system whose security mechanisms implement the model to withstand attacks. A system is ultimately secure if no malicious principal, perhaps acting in collusion with others, can gain access to a resource. In general, the system is measured against a set of profiles $\mathcal{E}_{\mathcal{P}}$.

Evaluation requires a that a set of scenarios and potential weaknesses of the security model be considered. In particular:

1. *Behaviour Under Attack*. This concerns the robustness of the model under a specified class of attacks from malicious or malfunctioning principals. This issue quantifies the circumstances where malicious principals gain access to resources that normally, they should not be allowed to gain access.
2. *Human Error*. It is well-accepted that the human is the weak link in a computer system's security. For instance, the *application policy developer* may specify an incomplete policy, or the *security administrator* may use a policy or mechanism in an incorrect manner. This aspect seeks to measure the effects of policies that are incorrectly specified.

Behaviour Under Attack Concerning *Behaviour Under Attack*, there are a number of relevant attack scenarios to evaluate in SECURE. Our goal is to understand how the SECURE model behaves in the presence of these attacks. The measure of attack resistance or *robustness* is how the attack contributes to an increased probability of a malicious principal being given access to a resource (compared to the absence of attack).

The important attacks to consider relate to the fact that SECURE is a collaborative model, as principals depend heavily on others, e.g., for recommendations and trust referencing. The following list contains key attacks that need to be analysed in the context of SECURE.

– **Collusion Attacks**. These attacks involve the collaboration of a number of malicious principals. An example is one where several principals collude by making positive recommendations for each other. The aim of this attack is to encourage the attacked principal to raise its trust value for other principals so that this trust can be later exploited or abused. The opposite attack – the *defamation attack* – is also possible, where several principals collude to destroy another's reputation.
  In a *denial of service* attack, a principal is starved of the evidence it needs to make meaningful access control decisions. A related version of this attack involves overloading a principal with evidence or access requests, to the point that it becomes infeasible for the principal to handle all of the evidence.
– **Masquerading**. This class of attack relates to the misinterpretation of a malicious principal's identity. An example is the *Sybil* attack [9], where a malicious principal manages to generate false identities. If false identities can be easily generated – and

believed – a principal can be easily fooled. A related attack is *Identity Theft* where a principal manages to obtain the information needed to have it recognised as another principal. An identity attack can follow a *privacy* attack where the behavioural profile of a principal is learned.

– **Single Principal Attacks**. This is a miscellaneous grouping of attacks that deal with the conduct of a single principal. One attack is the *waiting attack*, where a malicious principal acts correctly for $\mathcal{X}$ number of interactions, thus having its trust value increase, in order to abuse its high trust value on the $\mathcal{X}+1$th interaction. A variant of this is the *oscillation attack*, where a principal varies its behaviour so that it is never "bad" enough to get ejected from the system. More generally, we would like to model the *random failure* and behaviour of a principal.

This description contains a wide range of scenarios. However, some attacks will be more important in the context of specific applications. For instance, defamation attacks are especially critical in reputation-based systems.

Human Error The evaluation issue *Human Error* is particularly relevant for the risk analysis component of SECURE. Experience with prototyping applications and their policies in SECURE to date suggests that this is the hardest policy element to design. This is understandable since it is the least typical computer security element of the framework. In the context of our evaluation, it is important to be able to measure the effects of incorrect cost functions, i.e., cost functions that are too strict or too lax.

Another policy element that influences the security of the system is Entity Recognition, since the policy developer is obliged to assign a *level of confidence* value to the outcome of this process, which is taken into account in the risk analysis. Again, it is important to understand the effects on access control when recognition is too weak or too strong, or when a level of confidence is inappropriate. The strength of the recognition system is its ability to distinguish principals. A weak system identifies a range of possible principals for an interaction (e.g., not enough evidence available). A strong system fails to identify any due to the uncertainty in the system and recognition mechanisms.

3 Evaluation of SECURE Components

The goal of this section is to identify evaluation criteria for SECURE. As suggested in Section 2.1, this requires that the impact be measured that each component has on the decision making process.

The major components of the SECURE model are illustrated in Figure 3. In the remainder of this section, we clarify the role of each major component on the decision making process. This clarification is important for the evaluation that follows later. It is not our intention here to make a detailed evaluation of the components; this can be found in other deliverables.

A principal consults a running version of SECURE when it receives a request for an action a from another principal $\mathcal{P}$. There are three phases in the decision process. The first is to attribute a trust value to $\mathcal{P}$ for a. The second is *risk assessment*. Each action outcome has a pre-defined cost function associated with it. For instance, the cost of

marking a valid message as SPAM has a higher cost associated than the cost of letting a SPAM message pass. The trust value of $\mathcal{P}$ is then applied to the cost functions for all outcomes of the action to give an *estimated cost*. In the SPAM filter for instance, if the trust value for $\mathcal{P}$ indicates that $\mathcal{P}$ has sent a lot of SPAM, then the estimated cost of accepting the message is close to the pre-defined costs of accepting SPAM messages. The third phase of the decision process is the access control policy. This is a threshold type policy: it considers the estimated cost of undertaking the action and then makes the final decision.

To model trust formation and evolution, principals can exchange *recommendations*. Trust also evolves by observing the outcome of an action. Recommendations and observations are collectively known as *evidence*. The trust life-cycle manager is responsible for producing new trust values based on accumulated evidence. The other major element of the framework is entity recognition; its role is to assign principal identifiers to requesting entities.

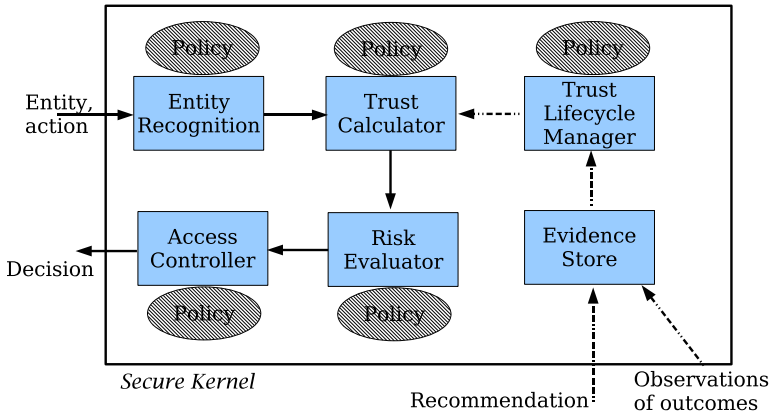

Fig. 3. The SECURE Model Components.

3.1 Risk Evaluator

The risk evaluation phase of SECURE applies a pre-defined cost function for an action to a request for that action. Consider an e-purse, where a request for a payment action is made for the amount m. One cost function (policy) might by $\alpha.m$, where α is a weight, meaning that the cost (of risk) is directly proportional to the amount of money transferred. In the overall decision framework, the importance of trust is that it determines the amount of risk that one is willing to take; in this example, this translates to the amount of payment [7].

In contrast to trust which may be considered to be globally computed, risk evaluation is a local process in which a principal assesses its personal exposure that results

from trusting another principal. In this way, a principal may make a local evaluation of global information (trust) to ensure that the views of others cannot contravene its own security policy, or force it to take a decision that it would not otherwise wish to make.

Risk is defined in terms of likelihood and impact, and the method of evaluation is outcome-based analysis [33]. If the analysis is static, that is, pre-computed and does not evolve throughout the lifetime of the principal, then it effectively forms part of the principal's *policy* and the key aspects that should be evaluated are the expressiveness, completeness, computational efficiency and ease-of-use for the policy-writer.

However, given the dynamic nature of trust, the SECURE consortium have also proposed that risk should evolve to take account of changing environments, contexts and circumstances [8]. Such a dynamic risk evaluator would need to be evaluated using quite different criteria, namely the rate of adaption to changes in context and circumstances and, as with the trust model, appropriate attack scenarios must be constructed and the risk evaluator shown to respond in the expected manner.

Risk assessment seems like the component that is least traditional computer science. However, this apparent complexity is offset by the fact that users do make an implicit risk assessment even when using traditional ACLs. It is considered good security practice to allocate access rights following the *need-to-know* principle, which is an implicit risk assessment practice. Nonetheless, a complete security evaluation requires that we measure the impact of policies with different risk assessments.

3.2 Trust Lifecycle Management

The Trust Lifecycle Manager (TLM) provides trust functions that facilitate evidence processing, in terms of how trust is formed, how it evolves over time and how it is exploited in the decision making process. The TLM is a policy that deals with issues such as whether a received recommendation should be believed, and if it is not totally believed, what alternative trust value should be assigned to the referenced principal. TLM also abstracts behaviour *patterns* from the reported behaviour of principals, and models how the decision maker's trust in others builds and erodes in the light of evidence.

The evaluation criteria of the TLM were demonstrated in the simulation of an e-purse [34]. The scenario involved an e-purse that a passenger uses to pay for a ticket on a bus (the decision making principal). In this scenario, a passenger's trustworthiness reflects the expected loss or gain in a transaction involving him. Our main aim was to measure how effective the TLM is at expressing how gathered evidence of a particular principal becomes a trust value for the decision maker, e.g., how a recommendation received – containing a trust value – gets turned into a local trust value. We conducted numerous experiments that use a wide variety of principals' behaviour profiles (i.e. norm behaviour, complex behaviour, and indiscernible or inconsistent behaviour) to verify that the TLM manages to run its functions on the accumulated evidence from previous interactions, and accordingly, give a better representation of the accurate trustworthiness of the principal in question. The simulation also focused on how the TLM abstracts the patterns of principal behaviour in their interactions to ease the decision making process for the bus company.

In the security evaluation, the impact of TLM through $\mathcal{E}_{\mathcal{P}}$ is measured by varying the manner in which received recommendations are translated into local trust values.

3.3 Entity Recognition

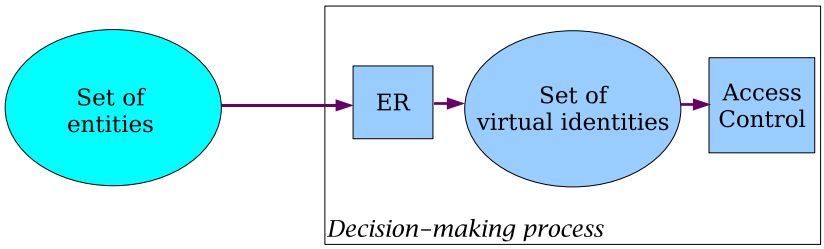

Fig. 4. ER maps interacting principals to virtual identities

The goal of Entity Recognition (ER), as suggested in Figure 4, is to bind a virtual identity to an interacting principal. When the ER component recognises a virtual identity, it means the entity has been met before; a new identity means that the entity has not previously been met. An attack on ER is one that leads to either premise being violated:

1. *Identity Usurption*. This occurs when a real-world identity obtains the information required to have itself identified as another. Common examples include password and private key stealing.
2. *Identity Manufacture*. This happens when a real-world identity can fabricate the data needed to have itself recognised as a new principal, and is also known as the Sybil attack [9]. An example of this is faked e-mail address used by a spammer. It is a particular problem in networks where identity management is decentralised or non-existent but where newcomers must be allowed.

Two key parameters need to be considered for ER evaluation. The first is whether the component relies on a centralised repository of information – as is the case with a public key system. A second parameter to consider is the information that a principal must furnish to have itself recognised. This is important in helping to judge the strength of the mechanism. For instance, IP addresses as ER information have proved to be fairly unreliable; DNA sampling on the other hand where real-world identities correspond to people is very reliable. Other factors to consider are privacy (whether the information furnished can be protected by the decision making principal), and responsibility (whether the link between the virtual identity and the real-world identity is strong enough for prosecution or insurance).

An example evaluation of the ER mechanism for SECURE is described in [29]. The outcome of this evaluation is a *confidence factor* ER_C. This is a percentage that indicates the level of confidence that the ER is able to accord to the virtual identity that it binds to the requesting principal. This value is then used in the security decision-making process.

With respect to a security framework evaluation, the ER evaluation must help us to answer the question: *what is the impact of an ER attack on the decision-making process?* The most likely way for an attacker to exploit ER attacks is *spoofing*, i.e., have itself recognised as an honest principal, either during an access request, or to create false recommendations for his buddies. For this reason, such collusion attacks should be part of the profile $\mathcal{E}_{\mathcal{P}}$.

4 The SPAM Example

In this section, we apply the methodology to a simulation of a SPAM filter for e-mail messages where a message gets classified as SPAM or as valid based on trust in the message sender. One reason for choosing the SPAM-filter application is that it is an example application for a highly distributed environment, where there is no global agreement as to what is a good decision, i.e., whether a specific message is valid or not.

4.1 Experimental Set-up

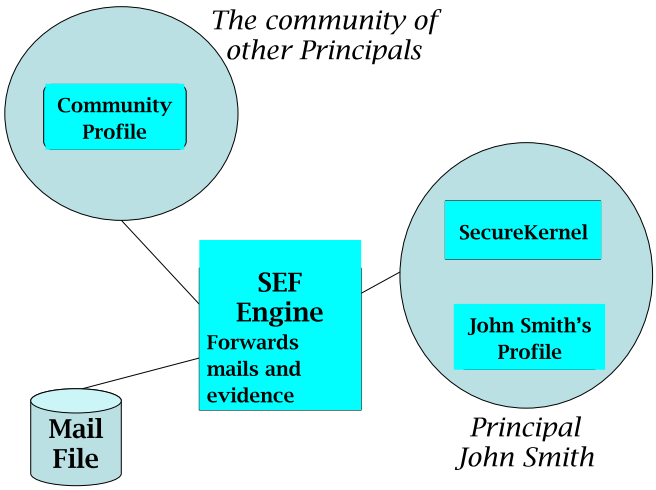

Fig. 5. The SECURE Evaluation Framework Configuration.

The experimental set-up is illustrated in Figure 5. The environment models and evaluates mail messages sent to principal John Smith. The evaluation uses a fixed set of mail messages that are stored in the mail file. Each message is processed by the policies that are plugged into the SECURE kernel, where a classification is assigned. Each message is pre-tagged as SPAM or valid, and for each evaluation run profile, the

accuracy of the results – $\mathcal{P}$ and $\mathcal{N}$ – is obtained by comparing the calculated SECURE classification to the pre-assigned SPAM tags of the messages.

The community profile of $\mathcal{E}_{\mathcal{P}}$ being evaluated includes spoofing attacks, for sending SPAM and false recommendations. As mentioned in the previous section, the key attack scenarios of the profile concern the impact of false and insufficient recommendations as well as the impact of weak recognition.

The mail message file is compiled from the **SpamAssassin** benchmark [2]. We used the easy_ham and spam files, composed respectively of valid messages and SPAM messages, and randomly merged the contents into the benchmark. There are 3051 messages in the benchmark, of which 2551 are valid and 500 are SPAM. The messages are sent from a community of 846 different senders. Of these, 425 senders are spammers, and 51 are repeat offenders (send more than 1 SPAM message). The number of valid mail senders is 421. Of these, 213 are once-off message senders and 208 send more than one message.

The remainder of this section presents three sets of validation measures for SECURE. These relate to three different components of SECURE: evidence management (recommendations), entity recognition and risk analysis. The measures seek to understand how failures of these components can impact on the decision making process of SECURE. In the case of risk analysis, we also measure the impact of human error.

4.2 Observations and Recommendations

Figure 6 shows the results for two profiles. There are four rows: the first row gives the number of correct classifications, and the second gives the percentage. The third and fourth rows give the breakdown on false positives and negatives respectively. The left column shows what happens when no SPAM filtering mechanism is used. Obviously, all SPAMs arrive in the Inbox, so the number of false negatives equals the number of SPAM messages in the benchmark. The second column considers the case where no recommendations are received, and where John Smith relies on his own observations. The most natural policy is to classify as SPAM any message received from a sender that previously sent SPAM – this is the learning effect brought by observations. For messages received from first time senders, the message will be classified as valid since no previous experience exists.

	No Filtering	**Observations Only**
Valid Classifications	2551	2626
Percentage	71.84%	86.07%
False Positive (FP)	0	0
False Negative (FN)	500	425

Fig. 6. Results when no filter is used, and when observations of previous spammers are taken into account.

The main benefit of a trust framework is the exchange of evidence, though the evidence can be falsified. In Figures 7 and 8 we model the effect of recommendations on the result. We simulate the arrival of recommendations for all principals. Each column deals with a different truth factor **t** for recommendations, from 0 (all recommendations are false) in the left column to 1 in the right-most column (all recommendations are true). The factor increases by 0.1 in each column. The right-most column of Figure 7 resembles trust stability since all past behaviour is known, so we detect all spammers. The policy used by John Smith is to consult recommendations when he has no experience concerning an email sender: he prefers observations to recommendations. This explains why the validity is not so bad when all recommendations sent are false, i.e., his own observations act as a safety net against poor recommendations. The results also illustrate the importance of buddy principals in whom John Smith can trust for good recommendations.

t	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
V	2205	2290	2375	2460	2545	2630	2715	2799	2883	2967	3051
%	72.27	75.06	77.84	80.63	83.42	86.2	88.99	91.74	94.49	97.25	100
FP	421	378	336	294	252	210	168	126	84	42	0
FN	425	383	340	297	254	211	168	126	84	42	0

Fig. 7. The correctness of the recommendations increase incrementally by 0.1.

Key Lesson Observations act as a safety net against poor recommendations. A principal believes what he observes more than what it hears from third parties.

4.3 Impact of Entity Recognition

The result of the security decision is heavily dependent on the entity recognition confidence factor ER_C. John Smith is in a quandary if his ER component recognises a principal p but with a low confidence value. In such a case, he might not even be able to use his trust value for p.

We can expect two kinds of behaviour for John Smith with respect to ER_C. If the calculated factor is low, then he might assume that the ER is unable to recognise the principal, and simply classifies the message as SPAM or as valid. These two cases are termed Easy Reject (ER) and Easy Accept (EA) respectively, and we evaluated with threshold ER_C factors of 25%, 50%, 75% and 100%. An alternative approach is to take product of the trust value for the recognised p and the ER_C factor for p, and to classify a message if a threshold has been reached. If the threshold is not reached, the message can be classified as SPAM or as valid. We name these cases Statistical Reject (SR) and Statistical Accept (SA) respectively. We evaluated for threshold values of 25%, 50%, 75% and 100%. In all experiments, the messages received have a confidence factor randomly distributed between 10% and 100%. Results are shown in Figures 9 and 10. These confirm that the best decisions are made when both trust in a principal's

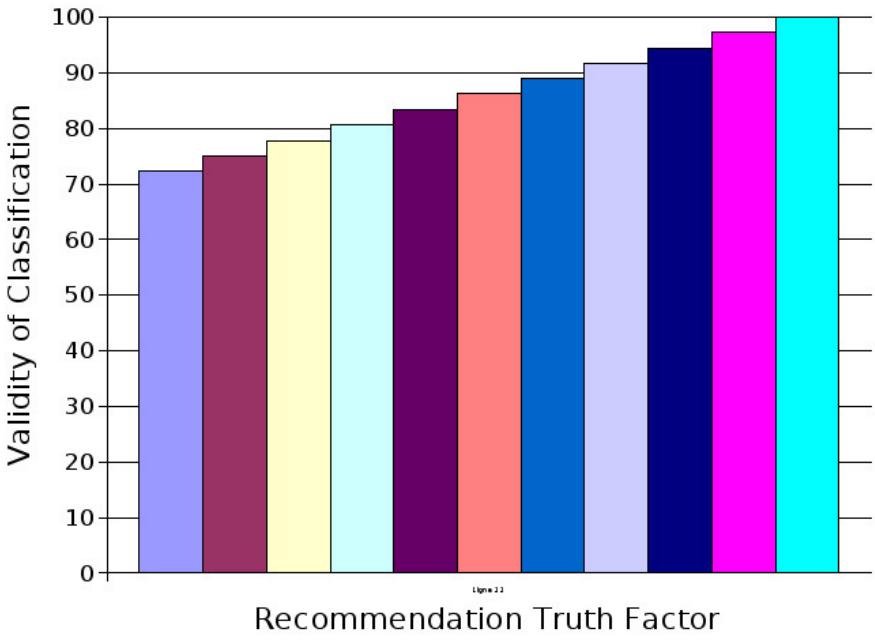

Fig. 8. Effect of recommendations - ranging from truth factors of 0 to 1 in units of 0.1.

recommendations and ER confidence values are used. The reason is that even if the ER_C factor is low, the possibility of a useful recommendation can help to avoid a valid message being classified as SPAM.

Tol	**25**	**50**	**75**	**100**	**25**	**50**	**75**	**100**
V	2539	2030	1252	742	2952	2851	2714	2614
%	83.22	66.54	41.04	24.32	96.76	93.44	88.95	85.68
FP	512	1021	1799	2309	0	0	0	0
FN	0	0	0	0	99	200	337	437
V	2903	2757	2513	2344	2958	2855	2719	2627
%	95.15	90.36	82.37	76.83	96.95	93.58	89.12	86.10
FP	148	294	538	707	0	0	0	0
FN	0	0	0	0	93	196	332	424

Fig. 9. The table is divided into four parts. The top left part models the Easy Reject (ER) policy and the top rightmost the Easy Accept (EA). The bottom left and right sub-tables implement the Statistical Reject (SR) and Statistical Accept (SA) policies.

Key Lesson A reliable friend (recommender) has a significantly positive impact on the decision making process. It is worth taking a friend's advice despite weaknesses in the entity recognition component.

4.4 Impact of Risk Modelling

As mentioned, the risk policy component is the least traditional of the SECURE components, so it is worth examining the impact of this component on the decision making process.

In the following evaluation, we use the risk policy described in Deliverable 3.2, entitled *Trust-based Access Control Model*. The cost function, P, used for risk evaluation is the cost of classifying a valid mail message as SPAM. It is defined as a function of trust and the Bayesian evaluation of the message:

$$\text{P defined as } (q * t) + (1 - q * \mathsf{bayes})$$

Both t and bayes are probabilities; t is the proportion of positive experiences to total experiences, the latter is got by a Bayesian analysis. The weight factor q is based on confidence in the entity recognition value and previous history (number of positive and negative experiences) with the interacting principal. The aim is thus to rely on trust information as long as this is available. The policy defines q as

$$\text{ER}_C \times (\, 1 - e^{-history/H})$$

where ER_C is the confidence in the entity recognition, history is the number of previous interactions with the recognised principal, and H is a weighting factor.

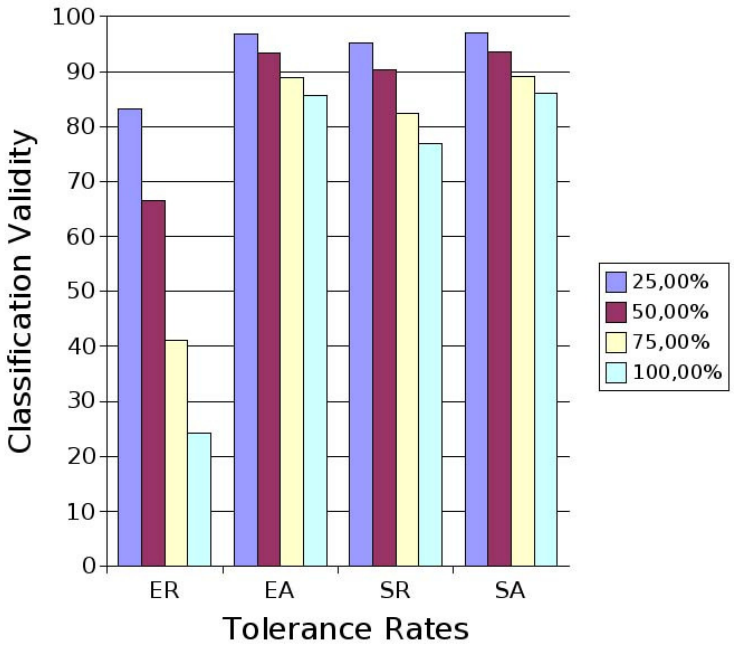

Fig. 10. Different treatments of Entity Recognition Confidence.

The risk policy assigns cost E to a valid mail being wrongly classified and benefit -1 to a SPAM message being wrongly classified. The risk function calculates (P $\times$ E + (-1) $\times$ (1-P)), which is equivalent to (P $\times$ (E + 1) -1). If this value is positive, then the cost of SPAM dominates, and the access control policy rejects the message. Otherwise, the message is classed as valid. Table 11 and Figure 12 plot the results for the simulation run with values for E from 0 to 6. The value bayes was set to 0.5 in the tests in order to concentrate on the impact of the trust part of the cost calculations; H was set to 2.5.

E	**0**	**1**	**2**	**3**	**4**	**5**	**6**
V	500	2626	2575	2562	2561	2558	2551
%	16.39	86.07	84.39	83.97	83.94	83.84	83.61
FP	2551	0	0	0	0	0	0
FN	0	425	476	489	490	493	500

Fig. 11. The table shows the evolution of validity as E goes from 0 to 6.

There are several points of note in these results. First, when the cost of classifying valid messages as SPAM is 0, then the benefit of a classifying a SPAM message as SPAM outweighs false positives. The policy thus errs on the side of classifying messages as SPAM. In other words, when in doubt, have the message classed as SPAM. The validity is very low – 16.39% – as most messages in the benchmark are valid messages. A second observation is that the best results are got when E is 1, and that they progressively diminish as E increases. This deterioration is explained by an increase in false negatives. As the cost of false positives increases, the policy errs more on the safe side – which means allowing more SPAM messages to be classed as valid. Third, it is interesting to note that the variation in validity is weak for E $>$ 1; this is explained by the reliance on the history information in the calculation. Thus, like for recommendations, a principal's observations act as a safety net against recurring spammers.

Key Lesson The risk component is crucial when deciding on which side of the yes/no decision process the SECURE kernel should be tuned.

5 Model to Implementation

The measures of the preceding section capture the model's effectiveness in making the right decisions in a decision-making process. This is a primary measure of a model, but there is another: the feasibility of the model's implementation in real systems. A model that cannot be easily implemented cannot be used, so serves no purpose.

For SECURE, implementation is important for two reasons. First, devices of autonomous systems today are rarely equipped with trusted hardware, despite recent and on-going efforts from the Trusted Computing Group [32]. Thus, instances of SECURE are currently in software and this makes the challenge more important. Second, the risk analysis component of the SECURE framework needs to take the strength of the

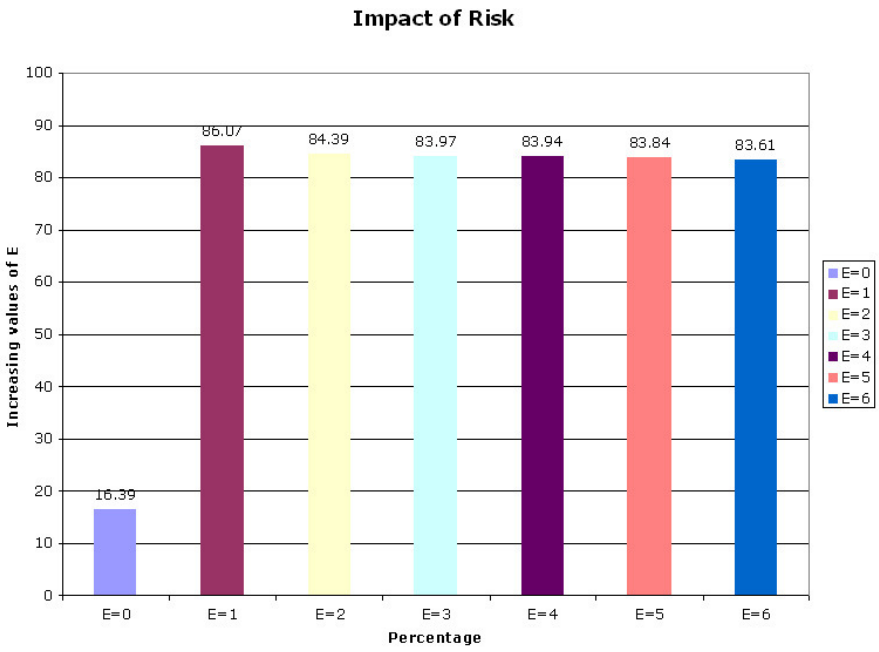

Fig. 12. Modelling different risk policies.

underlying implementation into account, as attacks on the implementation can undermine the model's secure operation. For this reason, we need an understanding of the implementation possibilities of the model.

We use the term *kernel* to denote the operational implementation of the SECURE model of a principal in a run-time system. Its role is to store all security data, e.g., evidence received and trust values, as well as the policy components that are consulted by the principal during the decision making process. Though the structure must mirror the components of the framework *model*, the kernel is a *systems* component and its design reflects this.

This section outlines the main systems challenges to consider when implementing a kernel.

1) Representing Principals The meaning of a *principal* has evolved over the years in computer security. It has gone from meaning a user, to a user in a certain role (e.g., in Multics [25]), to a public key (e.g., in BAN [20]). In SECURE, it refers to any autonomous entity that can initiate an action, that is capable of taking a decision, and in which another principal may have to place trust for an action. At the implementation level, a principal is an execution entity with support for protecting its data from others.

A principal represents a unit of protection, and typically corresponds to a process in OS environments, or to a node (platform). Thus, the system is able to associate a storage area with the principal that is protected from access by other principals. Protection is needed so that a principal cannot steal or corrupt data belonging to another principal. Protection boundaries implemented by OS processes are limited to runtime data. The kernel architect must ensure that security data made persistent – on disk – also remains protected.

A principal at runtime must possess its own communication endpoint so that other principals, perhaps running on different nodes, can name it and communicate with it. This task is becoming simpler in autonomous systems since, despite the variety of networks, socket stacks now exist for Bluetooth [11] and WIFI. However, the kernel architect must still ensure that a principal process does not get a socket receive handle on the socket port used by another principal.

The design choices must be made explicit by the kernel architect, as they are used later when evaluating the *Technical Trust* (Point 3). In the case of SECURE, a principal is instantiated as a Java Virtual Machine (JVM). All standard OS run a JVM in its own process, so several principals can be instantiated on a single node. The communication and persistence components are coded by the application developer.

2) Protecting the Trust Engines Due to autonomy, the kernel must be implemented on the same node as the principal, or on a node that is guaranteed to be accessible and trustworthy to the principal at all times. Security concerns exist even when the kernel is locally implemented since nodes can import library code that originates from an unknown or untrustworthy source. The challenge is twofold:

1. *Tamper-proof.* Software that runs on the node must not be able to interfere with the correct functioning of the kernel.

2. *Trusted Path*. When a principal's application code invokes the kernel to make a security decision, it must be certain that it is talking to the real kernel and not to malware masquerading as a kernel that an attacker has inserted into the call path of the kernel.

In SECURE, we built the kernel in software using a set of engineering principles that favour the tamper-proofness and trusted path properties – sometimes at the expense of API facility and performance. These principles are summarised in the appendices. However, they only make it harder for a malicious user to corrupt a kernel, not impossible. A measure of technical trust is therefore required.

3) Technical Trust The fact the nodes are not under the control of a universally trusted authority means that their kernels can be subverted by malicious users. It is therefore possible that a principal collaborates with an entity that has been compromised, and whose evidence and messages sent are completely false.

One way to address this issue is to make the risk of interacting with a compromised principal explicit in the decision making process. This is one reason why the SECURE model includes a risk evaluation component. It is also the reason why the design choices of the *Representing Principals* and *Protecting the Trust Engines* points must be made explicit, so that a meaningful risk analysis can be made.

The entity recognition module includes another example of this. We identified layers of trust which can be divided into two main categories: trust in the underlying technology and trust between entities. These layers together form end-to-end trust. In previous work [26], we presented how trust in different entity recognition schemes based on the average attack space [31] and other approaches can be estimated. It is the reason that the recognition process in SECURE includes, as output, a level of confidence in recognition. In the SECURE API, this is represented by the class `EntityRecogniser.Confidence`. This value is fed to the risk evaluator in the decision making process – via the RecognisedPrincipal object – where a decision can be made whether sufficient technical trust is established to take the risk of granting access.

4) Lack of Global Naming There need be no uniform naming scheme in an autonomous system. This impacts on principal naming and subsequent identification, as well as their binding to real-world principals. In the Simple Distributed Security Infrastructure SDSI [19] – an architecture designed for the Internet – two different principals may link their local name spaces with compound names. This allows Principal A to name a principal as B'C, meaning the principal that Principal B, in the name space of A, refers to by the name C. As a result, a principal may be known to two other principals by two different names, which complicates the treatment of evidence.

In SECURE, whenever a principal wants to refer to another principal $\mathcal{P}$, it provides *recognition clues* for other principals to recognise $\mathcal{P}$. Other principals interpret these clues in their own way, depending on the entity recognition component employed. For example, we developed a vision-based entity recognition (VER [28]) scheme where the clues consist of sequences of images of people passing in front of cameras. In this case, the principals carry out their own recognition process based on the images provided. There is a risk that recognition processes of two different principals recognise different principals for the same images; this depends on the security

strength and nature of the recognition scheme. This is another reason why technical trust is modelled in the entity recognition process. In the SECURE API, recognition clues are represented by the class `PrincipalInformation`; principals are represented by the class `RecognisedPrincipal`. Only instances of the former class are actually exchanged between principals (e.g., in recommendations) as each principal's recognition process constructs `RecognisedPrincipal` objects from the furnished `PrincipalInformation`.

5) Portability As with any system development, the ability of the system to run on different hardware and OS environments is considered an advantage. From a cost perspective, it is preferable that a software solution be adopted for this. In SECURE, this was our reason for implementing the kernel in the Java programming language [3].

However, the notion of portability extends to applications and its policies. That is, it should be possible to reuse the same kernel for different applications, and even for the same application but with different policies, e.g., to enable each principal to employ its own trust and access control policies. This means that the kernel contain a policy neutral *kernel* and that components like access control and entity recognition be pluggable.

Obtaining portability in practice requires that non-security features be integrated into the kernel. In any system, the three key features that can vary from platform to platform are storage, network and logging. These are relevant to SECURE for persistence of principal data, exchange of evidence, and monitoring respectively. This means that an application policy developer must not only specify security policies, but also specify how data is stored and exchanged. This explains the diversity of components in Figure 13.

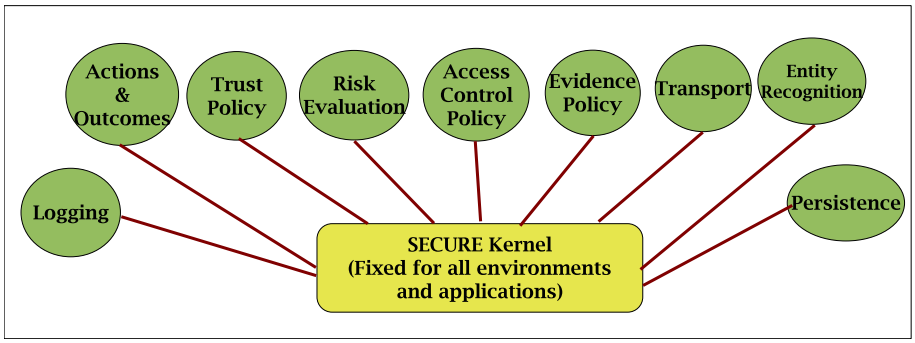

Fig. 13. A kernel runs several pluggable modules.

6) Layers of Trust Policy code is application and platform-specific, and is therefore written by the application security developer. Typically, the application software trusts the policy code. On the other hand, the trust engine places minimum trust in this code because it is not necessarily written by security experts: the kernel must be robust

against all independently written code. This explains the protection barrier line that separates the trust engine from all other software in Figure 14. Lastly, the application code is not trusted at all by the trust engine. It is not necessarily because the trust engine considers each user as malicious, but application software can contain errors that lead to security holes. Further, modern environments increasingly rely on foreign code, e.g., plug-ins and applets, that can originate from untrusted nodes. Software architectures are poorly adapted to this hierarchal layering, as they generally tend to favour total vertical protection barriers between programs, e.g., [4, 16, 25].

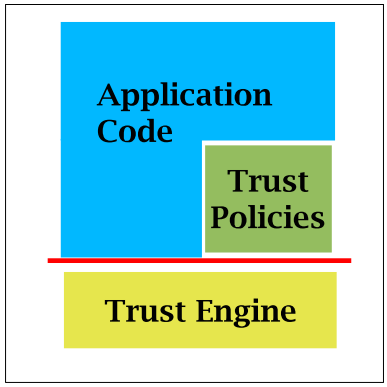

Fig. 14. Three levels of technical trust for a node's software.

6 An API for SECURE

This section describes an implementation of SECURE, and its use for the SPAM filter application. We begin the section with an overview of the SECURE API. We then look at how this API was applied for a SPAM filtering application. We close in Section 6.3 with a presentation of the SECURE Evaluation Framework. The complete SECURE code base contains more than 10.000 lines of Java code.

A series of engineering principles that we applied to the design are presented in the appendices. Since we programmed in the Java language, these are very Java-specific. Their importance is that they aim to respond to the challenge of ensuring the properties of tamper-proofness and trusted path, as well as the three levels of trusted software illustrated in Figure 14.

6.1 Overview of the SECURE API

We programmed the trust engine using the Java language [3] because of its type-safety and availability of its runtime environment (JRE) on a range of platforms. The kernel contains around 7000 lines of code. Figure 15 lists some classes from the API. The

classes correspond to the key abstractions of the framework. The main class is SecureKernel whose decide method is invoked by a principal's application code whenever a decision needs to be taken by the application. The trust engine implements a *consultative mediation* approach, in much the same way that the current Java security model works, in that an application explicitly invokes the kernel for a decision, e.g.,

```
public MailMessage fetch(User name, Password passwd) {
    SecureKernel kernel = SecureKernel.getSecureKernel();
    MailMessage message = server.getNextMessage(name, passwd);
    PrincipalInformation pi = new PrincipalInformation(message.getSender());
    if ( kernel.decide(ReceiveAction, pi, message) == MessageIsSpam )
        message.addHeader("X-Spam-Value", "Spam");
    else
        message.addHeader("X-Spam-Value", "SECURE-OK");
    return message;
}
```

To use the API, the application developer defines the set of policies for trust, risk, access control, etc. This done by coding by coding an Action (e.g., ReceiveAction in the SPAM filter), in which one specifies the Action.Outcomes and OutcomeCosts, and then coding the appropriate subclasses of TrustLifecycleManager, AccessController, etc. to represent the trust, access control policies, etc. A class must be coded for each of the components plugged into the kernel, as suggested in Figure 13. Appendix B gives the code extract from the SPAM filter application that sets up the SECURE policies: the purpose of showing this code extract is only to give an idea of how developers actually use the API.

6.2 SPAM Filter Case Study

The structure of the application is illustrated in Figure 16. A user – John Smith – has a mail client that is configured to use an IMAP and SMTP proxy. The role of the proxy is to interpose on messages received from the server, and to declare the message as SPAM or as valid, based on a SECURE decision. Each SPAM message detected by the SECURE framework is marked. This is done through the addition of a message header field **X-Spam-Value**. The mail client employs a filter to send SPAM messages to a special folder.

In this application, a principal represents a mail user, or more specifically, an e-mail address. A mail proxy runs as a separate process, on a machine that is specified by the mail client configuration. In the case of a false positive or false negative, the user can move the message from or to the SPAM folder. This move request is intercepted by the proxy and interpreted as an observation of an outcome (of a message being SPAM or valid). This is why the SPAM filter acts as a proxy on the IMAP protocol – used by clients to download messages from the mail server – since this protocol allows clients to create mail folders on the server and to copy messages between them. The POP3 protocol, also used by mail clients for downloading messages, does not have this possibility, so the best one could do is simply to mark the messages. All standard mail clients today can be configured to use IMAP and SMTP proxies, i.e., the open source Mozilla clients, Netscape, as well as MicroSoft's Outlook (Express). Thus, the SECURE filter is client-independent. The code that processes the IMAP commands and implements the policy components contains around 3000 lines of pure Java code.

```
public final class SecureKernel {
	public Decision decide(Action, PrincipalInformation, Object[]);
	public void registerEvidence(Evidence);
	public static SecureKernel initialise(AccessController, Transport, EvidenceManagement, ....);
}

public final class Action {
	public static Action defineAction(String, Eventstructure);
	public TrustDomain getTrustDomain();
	public Outcome getOutcome(String);
	public Outcome[] getAllOutcomes();
}

public final class Outcome {
	public OutcomeCosts getOutcomeCosts();
}

public final class TrustDomain {
	public TrustValue getBottom();
	public TrustValue valueOf(TrustValue, Outcome, Triple);
}

public final class TrustValue {
	public boolean isLessTrustedThan(TrustValue, Outcome);
	public boolean isLessPreciseThan(TrustValue, Outcome);
}

public abstract class EntityRecognition {
	protected final RecognisedPrincipal createPrincipal(PrincipalInformation, Confidence);
	public abstract RecognisedPrincipal recognise(PrincipalInformation, Object);
}

public final class RecognisedPrincipal {
	protected final TrustValue trustInPrincipal(RecognisedPrincipal);
	public PrincipalInformation getPrincipalInformation();
	public EntityRecognition.Confidence getConfidence(); // ER confidence level
}

public final class Recommendation {
	public Recommendation(PrincipalInformation, PrincipalInformation, TrustValue);
	public PrincipalInformation getReferee();
}

public abstract class AccessController {
	public abstract Decision isAllowed(Action.Execution, TrustValue, RiskMetric);
}

public final class RiskEvaluator {
	public RiskMetric evaluateRisk(Action, RecognisedPrincipal, TrustValue);
}

public abstract class TrustLifecycleManager {
	private EvidenceStore; // For recommendations and observations
	protected abstract void update(Evidence);
	public final TrustValue trustIn(RecognisedPrincipal);
	public final void setTrust(RecognisedPrincipal, TrustValue);
}

public abstract class TransportLayer {
	private ListenerThread;
	public void distributeEvidence(Evidence, RecognisedPrincipal);
	public final TrustValue getTrustValue(RecognisedPrincipal, PrincipalInformation);
}
```

Fig. 15. Extract of SECURE API.

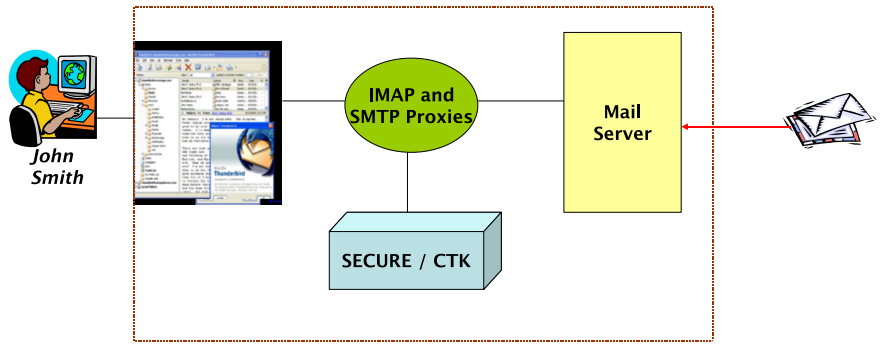

Fig. 16. The SECURE SPAM filter.

The following code extract is taken from the trust policy used by John Smith's proxy. The key method of the policy is update which is called for each item of evidence – observations and recommendations – registered with the SECURE kernel. The policy with respect to recommendations is to accept them, and consequently update the trust state, as long as the sender of the recommendation is one of John Smith's friends. The set of friends is fixed during application boot-time (see Appendix B). An observation is registered when a folder move is undertaken to or from the SPAM folder. The effect is to undo the classification, which means updating the trust value to reflect the correct outcome.

```
public class TrustPolicy extends TrustLifecycleManager {

    private TrustDomain domain;
    private Action action; // that is being controlled by the TLM
    private Logger log;

    public TrustPolicy(TrustDomain domain, Map mappings) {
        super(domain, mappings);
        this.domain = domain;
    }

    protected void update(Evidence ev) {
        if (ev instanceof Recommendation) {
            Recommendation r = (Recommendation) ev;
            Triple triple =
                r.getTrustValue().getTriple(
                        MailerAction.ReceiveMail.getOutcome("Valid"));

        if (Friends.isMember(r.getReferee()))
            setTrust(r.getSubject(), r.getTrustValue());

        if (ev instanceof OutcomeObservation) {
            action = MailerAction.ReceiveMail;
            OutcomeObservation obs = (OutcomeObservation) ev;
            // Match the destination folder to an outcome event
            String folder = obs.getFolder();
            Action.Outcome outcome = action.getOutcome("Valid");
            TrustValue oldT = trustIn(obs.getPrincipalInformation());
            Triple oldTriple = oldT.getTriple(outcome);
            Triple newTriple;
```

```
            if (folder.equalsIgnoreCase("Spam"))
                newTriple =
                        new Triple(
                            oldTriple.getS(),
                            oldTriple.getI(),
                            oldTriple.getC() - 1);
            else
                newTriple =
                    new Triple(
                            oldTriple.getS() + 1,
                            oldTriple.getI(),
                            oldTriple.getC());
            TrustValue newT = domain.valueOf(oldT, outcome, newTriple);
            setTrust(obs.getPrincipalInformation(), newT);
        }
    }
}
```

Entity Recognition is got by the Claim Tool Kit (CTK) [27], which provides message-based recognition. It is a reusable software tool kit of approximately 22000 lines of code with 30 core classes and 50 implementation example classes.

The main challenge for entity recognition was to allow users to easily and cheaply exchange email addresses without making it easy to spoof addresses. A check needs to be made that the message comes from the email account specified by its 'From:' address. For this, both the receiver's and sender's CTK SMTP proxy store customised hashes of previous emails exchanged between the two, and are able to bootstrap each other using an augmented challenge/response technique. This is more elaborate than a challenge/response mechanism that uses the 'From:' header as an authentication key, as it relies on ownership of email accounts and proof of possession of a shared history (previous messages). By presenting the correct hashes, the sender proves that he has access to previous messages sent, and hence, is the originating email account. This proves that a spoofing attack is not happening. A stronger scheme is also possible (and recommended), where each CTK is given a key pair to sign exchanged messages. Recognition is sufficient to build trust based on pieces of evidence such as direct observations and recommendations. There is no need to authenticate the real-world identity with a certified binding to the public key, so registration is cheap, fast and convenient for users. This is in contrast to public key webs of trust schemes or ones requiring manual enrolment.

6.3 The SECURE Evaluation Framework

Clearly, SECURE policies have a large number of parameters that can influence the outcome of the decision process. A developer who employs SECURE for his application may wish to fine-tune his policies before application deployment. For this reason, we believe that the evaluation framework that we developed for validation should be included as part of the SECURE code base.

A list of the main classes of the SECURE Evaluation Framework (package sef.framework) is given in Figure 17. A simulation run requires that CommunityProfile and PolicyProfile objects be defined and plugged in. The community profile can be defined by extending the CommunityProfile class, or by using the AbstractCommunityProfile which creates a community profile from a configuration file. This file defines the

recommendations sent to John Smith as well as the replies to reference requests from Smith. As illustration, the following lines are taken from a community profile configuration file:

```
REFERENCES
#Format: Referee Candidate TrustValue
admin@cui.unige.ch *@cui.unige.ch [10, 0, 0]

RECOMMENDATIONS
# Format: event number, sender of recommendation, triple value.
3 vinny.cahill@cs.tcd.ie fiachra@gonzaga.ie [10, 0, 0]
6 Tony.Blair@ds Nathan.Dimmock@cl.cam.ac.uk [10, 0, 0]
```

The simulation is centred around the sef.framework.Engine class that reads the profiles, a mail message benchmark file and that simulates the sending of messages and recommendations to JohnSmith (using the sendMail and sendRecommendation methods respectively). The engine queries the community profile object for trust references. It is important to note that despite the fact that the simulation is based on the SPAM filter application, there is no SMTP or IMAP processing. The simulation is thus independent of the SPAM application and can be used to evaluate a wider range of applications and policies.

7 Related Work

A quantitative approach for assessing the security properties of trust metrics is described in [33]; the authors describe an analytical evaluation approach, but only apply it to the graphs of trust relations. Our work in this paper completes this by assessing the security properties of the trust-based model as a whole by including aspects such as identity, trust life-cycle management and risk.

Massa and Bhattacharjee empirically analyse a recommendation system in [23]. This is a system where members can annotate items (e.g., movies, books, cars) with their personal recommendations. Trust is an important element of these systems since members believe a recommendation for an item based on their trust in the recommending user. The authors consider the weaknesses of recommendation systems with regards to sparseness (most items have zero or one review, and most members know very few others), cold start (the issue of how members can learn to trust each other) and vulnerability to fake reviews of items. The exchange of trust information can treat these all of these issues, mainly since the sparseness level drops. This happens because members can obtain trust information from others for any member who places a review in the system. Cold start users supply a friend's name. The mechanism is robust so long as fake users are not considered as friends.

Liu and Issarny present a recommendation distribution system and its evaluation in [22]. Recommendation is context-specific and evolves over time. The system is designed to overcome free-riders (principals that do not share recommendations), defamation attacks (propagation of incorrectly low recommendation values) and collusion attacks (propagation of incorrectly high recommendation values). The experiments indicate that the reputation adapts correctly to these attacks. Like for [23], the validation

```
public final class AbstractCommunityProfile extends CommunityProfile {
        ... }

public abstract class CommunityProfile {
        public MessageDescription getMessageDescription(int msgIndex);
        public TrustValue getReferencedTrustValue(PrincipalInformation referee, PrincipalInformation subject);
        public boolean hasMoreEvents();
        public SimulationEvent nextEvent();
}

public final class EvaluationResult {
        public void display();
        public int getNumFalsePositives();
        public int getNumFalseNegatives();
        public Profile getProfile();
}

public final class JohnSmith {
        public void sendMail(MessageDescription message);
        public void sendRecommendation(Recommendation rec);
}

public final class MailFile {
        public int getERConfidence(int msgIndex);
        public String getSender(String msgIndex);
        public String getSubject(int msgIndex);
        public boolean isSpam(int msgIndex);
        public String getMessageText(int msgIndex);
}

public final class Profile {
        PolicyProfile getPolicyProfile();
        CommunityProfile getCommunityProfile();
}

public final class PolicyProfile {
        public AccessController getAccessController();
        public EvidenceManagementPolicy getEvidenceManagementPolicy();
        public TrustLifecycleManager getTrustLifecycleManager();
}

public final class SimulationEvent {
        public boolean isMailMessage();
        public boolean isRecommendation();
        public Recommendation get Recommendation();
        public MailDescription getMessage();
}
```

Fig. 17. Extract of SECURE Evaluation Framework API.

criteria proposed include validity (ability to distinguish between honest and dishonest principals), timeliness or response, robustness under attacks and resource economical. The evaluation is thus with respect to a single component – aspects such as trust in principal recognition and policy expression. More importantly, the evaluation does not link the recommendation system to the strength of a security process that employs it. In fact, there is no notion of security in this system.

Other researchers have attempted to evaluate the mapping between trust model and actual human intuitions of trust by psychological experiment [15, 17]. Their results so far are very insightful concerning to how humans really do *trust* each other, but as yet are not detailed enough to really give much guidance to designers of computational trust systems. That said, it is important to avoid the assumption that a better understanding of the human trust process can lead to better security.

Since trust is an element of a chain built for security, it is important to consider how other work on security impacts on trust. For instance, the Trusted Computer Group[2] was founded in 2003 and is made up of nearly all major software and hardware platform vendors. The goal of the consortium is to define a standard for trusted platform computing. More precisely, the aim of the organisation is to produce a standard that allows any device to prove to another device that the software it runs has not been tampered with.

The first result of the Trusted Computer Group has been the definition of a hardware platform architecture – the *Trusted Computing Platform Architecture* – that can be included in all types of hardware devices [32]. This platform is supposed to be implemented with the aid of a hardware chip called the Trusted Platform Module (TPM). The platform takes measures of the running software and stores these in the TPM. The measures are secure hashes of code sequences. A device transmits its measures to anyone that wishes to verify the correctness of the device's software. The verifying party validates these measures with the aid of certificates that the software manufacturer issues for its software.

The work of the Trusted Computer Group is very promising, but it essentially remains a specification. Few computers today possess trusted hardware, so software techniques still need to be relied upon for security.

There has been considerable work on security techniques for Java platforms that can be exploited for tamper-proofness [4, 16]. A key example is isolates. An *isolate* defines a container in which to execute a Java program. The container guarantees the strict isolation of the program from all other programs executed as isolates: each isolate has its own copy of loaded classes, including core classes, so that isolates do not share static variables or any class states (e.g., initialization status, instance of `java.lang.Class` for the same class, etc.). Furthermore, access to an isolate is only via reference objects that it creates, i.e., there is no sharing between isolates. Isolate creation and life cycle management are handled by the Isolation API, the formal output of the JSR-121 [16].

The aim of isolation mechanisms is to enforce secure separation between mistrusting programs. This is one way of implementing the barrier between the kernel and the application space. Yet there remain situations where one would like to co-locate trusted

[2] www.trustedcomputinggroup.org

code with application code, or where the system is composed of components of different trust levels, e.g., the layers of trust in Figure 14.

8 Conclusions

This paper discussed the evaluation of the SECURE framework, along with a methodology for evaluating trust-based frameworks. We believe that it is important to distinguish the evaluation of security, from the evaluation of the trust subsystem and of other components, and to clearly delimit the role and functionality of each component. A meaningful analysis is otherwise not possible.

The key lessons learned from the evaluation of SECURE are the following:

- The key advantage of a trust-based security model is that it permits security decisions to evolve over time as evidence becomes known to principals. This means that each principal can take a decision based on his own experience in the system. This property is very important, since it can prevent a global attack on the system from succeeding.
- The main weakness of trust-based models is Assertion T: relying exclusively on evidence is still not enough to guarantee that a security mechanism built over the trust model combats all attacks.
- Many researchers of trust-based security systems define the success criteria of models with their proximity to human notions of trust. As such, more analytical measures of security have been overlooked. We have attempted to address this issue in the SECURE project.
- Bayesian and SECURE approaches to SPAM filtering are *complementary*. Each method addresses weaknesses that the other poses. The most effective SPAM filtering is got by combining both methods.
- SECURE can be implemented with a reasonable degree of security in software using a strongly typed programming language. More generally, an important validation measure for a security model in an autonomous system is the facility of its implementation, since security data cannot be stored on trustworthy third parties (as such parties do not exist).

Fundamentally, evaluation of the SECURE approach will necessitate experience with a number of applications. We have not had time in the context of this project to undertake such development. However, the initial results are positive, and we now have a small community of potential users of the code base, so work on SECURE does not stop with the EU project. This in itself is another positive validation result.

This work is sponsored by the European Union, which funded the IST-2001-32486 SECURE project.

References

1. Ross J. Anderson. *Security Engineering — A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2001.

2. Apache. Spamassassin, http://spamassassin.apache.org.
3. Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley Publishing Company, 1996.
4. Godmar Back, Jay Lepreau, Leigh Stoller, Patrick Tullmann, and Wilson C. Hsieh. Techniques for the design of java operating systems, April 26 2000.
5. Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. Technical Report 96-17, DIMACS, June 28 1996.
6. Joshua Bloch. *Effective Java*. The Java Series. Addison-Wesley, 2001.
7. V. Cahill and etal. Using Trust for Secure Collaboration in Uncertain Environments. In *IEEE Pervasive*, volume 2(3), pages 52–61. IEEE, April 2003.
8. Nathan Dimmock, Jean Bacon, David Ingram, and Ken Moody. Trust-based access control model. EU IST-FET SECURE Project Deliverable, September 2004.
9. John R. Douceur and Judith S. Donath. The Sybil Attack. February 22 2002.
10. Deborah Estrin, Ramesh Govindan, and John Heidemann. Embedding the Internet: Introduction. *Communications of the ACM*, 43(5):38–38, May 2000.
11. Kris Fleming, Uma Gadamsetty, Rajagopal Rajagopal, and Ramakesavan Ramakesavan. Architectural overview of Intel's Bluetooth software stack. *Intel Technology Journal*, (Q2):10, May 2000.
12. Hector Garcia-Molina, Mario T. Schlosser, and Sepandar D. Kamvar. The EigenTrust Algorithm for Reputation Management in P2P Networks, November 18 2002.
13. Li Gong. *Inside Java 2 Platform Security*. The Java Series. Addison Wesley, 1999.
14. Robert H. Istepanian, Swamy Laxminarayan, and Constantinos S. Pattichis. *M-Health: Emerging Mobile Health Systems*. Kluwer Academic/Plenum publishers, 2004.
15. Catholijn M. Jonker, Joost J. P. Schalken, Jan Theeuwes, and Jan Treur. Human Experiments in Trust Dynamics. In *Proceedings of the 2nd International Conference on Trust Management, LNCS 2995*, pages 206–220, March 2004.
16. JSR. Java System Requirements: Application Isolation. Technical Report 121, 2002.
17. Tim Kindberg, Abigail Sellen, and Erik Geelhoed. Security and Trust in Mobile Interactions: A Study of Users' Perceptions and Reasoning. In *Proceedings of the Sixth International Conference on Ubiquitous Computing (UbiComp 2004)*, 2004.
18. Butler Lampson and Ronald L. Rivest. SDSI – A Simple Distributed Security Infrastructure. Technical report, July 26 1996.
19. Butler Lampson and Ronald L. Rivest. SDSI - A Simple Distributed Security Infrastructure. October 1996.
20. Butler Lampson, Edward Wobber, Martin Abadi, and Mike Burrows. Authentication in the Taos Operating System. February 1994.
21. Marc Langheinrich. When Trust Does Not Compute — The Role of Trust in Ubiquitous Computing. *Workshop on Privacy at Ubicomp 2003*, 2201:1–8, 2003.
22. J. Liu and V. Issarny. Enhanced Reputation Mechanism for Mobile Ad Hoc Networks. In *Proceedings of the 2nd International Conference on Trust Management, LNCS 2995*, pages 48–62, March 2004.
23. Paolo Massa and Bobby Bhattacharjee. Using Trust in Recommender Systems: an Experimental Analysis. In *Proceedings of the 2nd International Conference on Trust Management, LNCS 2995*, pages 48–62, March 2004.
24. Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical Report HPL-2002-57R1, Hewlett Packard Laboratories, July 14 2003.
25. Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

26. J.-M. Seigneur, S. Farrell, C. D. Jensen, E. Gray, and Y. Chen. End-to-end Trust Starts with Recognition. In *Proceedings of the First International Conference on Security in Pervasive Computing, LNCS 2802, Springer-Verlag*, volume 280, March 2003.
27. J.-M. Seigneur and C. D. Jensen. The Claim Tool Kit for Ad-hoc Recognition of Peer Entities. In *Journal of Science of Computer Programming*. Elsevier, 2004.
28. J.-M. Seigneur, D. Solis, and F. Shevlin. Ambient Intelligence through Image Retrieval. In *Proceedings of the 3rd International Conference on Image and Video Retrieval, LNCS, Springer-Verlag*, March 2004.
29. Jean-Marc Seigneur, Nathan Dimmock, Ciaran Bryce, and Christian Jensen. Combating SPAM with Trustworthy Email Addresses. In *Proceedings of the 2nd International Conference on Privacy, Security and Trust*, pages 228–229, New Brunswick, June 9–19 2004. ACM Press.
30. Jean-Marc Seigneur and Christian Damsgaard Jensen. Privacy Recovery with Disposable Email Addresses. *IEEE Security & Privacy*, 1(6):35–39, November/December 2003.
31. Richard E. Smith. *Authentication: From Passwords to Public Keys*. Addison-Wesley, Reading, MA, USA, 2001.
32. Trusted-Computing-Group. TCG Main Specification. http://trustedcomputinggroup.org, August 2003. Version 1.1.
33. Andrew Twigg and Nathan Dimmock. Attack Resistance of Computational Trust Models. In *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises — Enterprise Security*, pages 281–282, June 2003.
34. Waleed Wagealla, Sotirios Terzis, Colin English, and Paddy Nixon. Simulation-based Assessment and Validation for the SECURE Collaboration Model. Technical report, University of Strathcylde, 2005.
35. Li Xiong and Ling Liu. A Reputation-based Trust Model for Peer-to-Peer Ecommerce Communities. In *Proceedings of the 4th ACM Conference on Electronic Commerce (EC-03)*, pages 228–229, New York, June 9–12 2003. ACM Press.
36. Nicola Zannone. A Survey on Trust Management Languages. Technical report, University of Verona, August 01 2004.

Appendix A. Java Security Engineering Principles

This appendix lists some of the concrete principles employed to ensure that the SECURE kernel could be implemented securely – in software. Use of these principles convinced us that a software solution is sufficient in many cases.

The notion of secure implementation here is the *trusted path* property. That is, a user program must be able to invoke the local SECURE kernel, and be sure that the kernel has not been compromised by another program, either running on the same machine or on another. In the latter case, corruption of the kernel could occur by an entity sending a virus or worm code unit.

The measures or principles used to achieve the implementation must be known in order to gauge the strength of the implementation. This, in turn can be used as a parameter to a principal's risk policy if the outcome of a decision really depends on whether a partner principal has been tampered with or not.

In most cases, the principles used just make good software engineering sense. However, they are not always applied by programmers. Our experience implementing SECURE has helped us to highlight the key principles that need to be applied to make a system, written in the Java programming language, secure.

Minimize Class Visibility The programmer must ensure that the declared scope of classes is as restricted as possible. Recall that in Java, a class can be declared as `public` or as package protected (no modifier). A public class $\mathcal{C}$ is visible to all application classes. If $\mathcal{C}$ is package protected, then it is only visible to other classes in the same package. This is an important feature because it means that even if a reference for an object of class $\mathcal{C}$ escapes into the application space, where this is a class representing a sensitive data structure, a cast to this object's class is prohibited if $\mathcal{C}$ is package protected. Thus, the application is unable to invoke sensitive methods on the object. In the SECURE API, the classes `EvidenceStore` and `TrustState` are package protected for this reason.

Control Object Modification Trust values need to be modified to represent new levels of trust. However, allowing application code to modify a trust value arbitrarily is clearly a security risk. There are two steps to the solution. The first is to make objects immutable, i.e., their state is fixed at the moment of their creation, and does not change. This has the advantage that even if undesirable aliasing arises on trust values in the evidence store, this cannot lead to modification attacks. The second element of the solution is to provide a factory method in the class to create instances of the object. Thus, in the case of `TrustValue`, when a new trust value needs to be created that is an increment of an existing value, then a new object is created using an `incS`, `incI` or `incC` method. I.e., the code of `incS` is return new TrustValue(t.s + 1, t.i, t.c).

Beware of Code Injection The major characteristic of object-oriented programming is inheritance coupled with sub-typing. This is the facility by which a class can be defined as an extension of an existing class. The new class may define new methods, or redefine methods of the original class. Sub-typing permits the behaviour of the system to be extended at runtime. Of course, this has a risk associated with it, since it opens a back door to the introduction of Trojan Horse code, and can lead to the *trusted path* property being broken, e.g., a corrupt sub-class of SecureKernel could break the whole of a principal's application. In the SECURE API, a majority of classes are final for this reason.

Control Object Creation The Java programming language uses the **new** operator to create an object, i.e., when executed, the expression `new C` creates an object of class `C`. This operation can be executed in any code section where this class is visible. However, object creation is an expensive operation in terms of time and space. The problem with the **new** operator is that it creates a new object – as long as there is sufficient heap memory available – without imposing any control on the creation.

The solution to this is to employ a factory method. This is a static method that creates an instance of the class, as was done in the previous example for TrustValue. There are several already known advantages to using factory methods [6]. For instance, a factory method need not create a new object, but rather, return a reference to an existing object. This is particularly useful when the class is immutable, because only a single instance of the class need ever be created; all client programs simply share the same instance.

From a security point of view there are several advantages. First, it is easier to enforce a trace on the objects created, e.g., to track the set of recommendation objects created by the platform. Second, it helps to a degree to counter denial of service attacks. I.e., the factory method can refuse to create new objects if some specified limit of objects have already been created. Of course, the application may attempt a denial of service attack on the local heap memory using classes in the application space, but at least the kernel itself cannot be exploited to launch these attacks. This is important since, for the application developer, the kernel is foreign code and he should have full confidence in its functioning. In SECURE, factory methods are used in several places, e.g., to create `TrustValue` and `RecognisedPrincipal` objects.

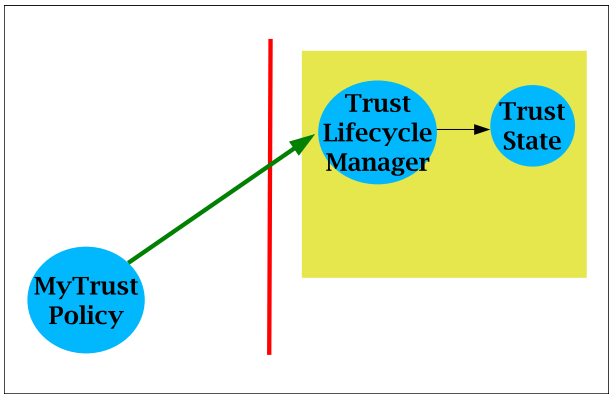

Fig. 18. Visibility across the protection barrier.

Keeping Scope Under Control An example of the three layers of security from a code perspective is given in Figure 18: a trust policy – `MyTrustPolicy` – in the application policy space which subclasses `TrustLifecycleManager`. An attribute of the latter class is `TrustState` which represents the mapping from principals to trust values in the kernel. In order to protect the trust state, we do not want the internals of this data structure to be visible or accessible in the application space, since this would allow malicious code to change trust value entries without the mediation of the Trust Lifecycle Management policy.

The security requirement is to ensure that the trust state can only be modified by policy code, i.e., from within the class `TrustLifecycleManager`. Further, we ensure that no references to the `TrustState` object can leak out from the policy code, c.f., Figure 18. This is achieved by judicious use of visibility modifiers. Note that the `state` field is declared as private, so it is inaccessible in subclasses, and thus in the application space. Access to the trust state from the policy code uses the `setTrust` and `trustIn` methods. These methods are final, so they cannot be redefined in malicious subclasses of `TrustLifecycleManager`. This explains why the setTrust and trustIn methods are protected.

Beware of Twigging Attacks When returning data from a method, a programmer has the choice of implementing a copy-by-reference or copy-by-value. When copy-by-reference is used, and the referenced object is mutable, then the caller may break encapsulation and modify the state of the attribute. Consider the class `Recommendation`. This has a `Date` attribute that represents an expiry date for the recommendation. If this field is returned by copy-by-reference, then the following code is possible. The solution is thus to employ copy by value as often as possible in the API.

```
Recommendation[] myRecs = Util.getRecommendationsForMe();
for (int i = 0; i < myRecs.length; i++) {
    Date d = myRecs[i].getExpirationDate();
    d.setYear(2010);    // Twigging Attack
}
```

Handle Exceptions The exception handling facility of the Java programming language is quite rich, and plays no small part in the success of the language. Its purpose is to allow programs to detect and recover from abnormal, though anticipated, situations. The main problem with exceptions is that they expand the scope of an object since they can carry objects with them as attributes that can be examined by client programs. This can obviously lead to a security hole. Thus, all exceptions thrown in the trust engine code must be caught and sanitised at the API boundary, i.e., for all methods in the public API, a check is made for exceptions, and a new exception thrown, i.e.,

```
public void APImethod() throws SecureException {
    try {
        .....
    } catch (Exception exc) { // Throw ``safe'' exception
        throw new SecureException(exc.toString()):
    }
}
```

Make the API Security Aware The restrictions imposed for security mostly make good software engineering sense, but in the case of SECURE at least, some choices have inconvenienced developers. Instances of the class `RecognisedPrincipal` can only be created by the SECURE kernel. Thus, similarly to the example above with `MyTrustPolicy`, the class `EntityRecogniser` contains final protected methods that generate these objects. Thus, the code that does the entity recognition for the application must be declared as a `EntityRecogniser` subclass. This meant a certain amount of restructuring of the CTK ER code in order to have it correctly – and securely – integrated into our framework. There is no quick technical solution to this example. However, it helps when the thinking behind the security aspects of the API is well explained in the API documentation, as third-party developers are less likely to be discouraged by the API. After all, security is a fundamental software quality, just as important as correctness and robustness. It is logical that the API reflect this.

Appendix B. Example API Usage

The following code extract is taken from the SPAM filter application proxy. The code sequence shows the initialisation of the Secure policies; this sequence executes before the proxy connects to the SMTP and IMAP servers.

```
public final class KernelSetUp {

    private static KernelSetUp setUp = null;
    private SecureKernel secureKernel;

    private KernelSetUp() {
        try {
            // Principals that we recognise
            PrincipalInformation self, admin, vinny, liz, waleed, jm, cui, fred;
            self = new PrincipalInformation("smith@localhost");
            fred = new PrincipalInformation("fred@localhost");
            admin = new PrincipalInformation("admin@cui.unige.ch");
            vinny = new PrincipalInformation("vinny.cahill@cs.tcd.ie");
            liz = new PrincipalInformation("liz.gray@cs.tcd.ie");
            waleed = new PrincipalInformation("waleed.wagealla@cis.strath.ac.uk");
            jm = new PrincipalInformation("jean-marc.seigneur@cs.tcd.ie");
            cui = new PrincipalInformation("*@cui.unige.ch");

            Friends.register(new PrincipalInformation[] {self, admin,
                        vinny, liz, waleed, jm, cui, fred });
            // Entity Recognition
            EntityRecognition er = new SimpleEntityRecognition(self, 100);

            // Event Structure
            HashSet events = new HashSet();
            HashMap causal = new HashMap();
            HashMap conflict = new HashMap();
            events.add("m_real"); events.add("m_spam");
            events.add("u_real"); events.add("u_spam");
            // Conflict graph
            HashSet mRealConflict = new HashSet();
            mRealConflict.add("m_spam"); mRealConflict.add("u_real");
            mRealConflict.add("u_spam");
            conflict.put("m_real", mRealConflict);
            HashSet mSpamConflict = new HashSet();
            mSpamConflict.add("m_real"); mSpamConflict.add("u_real");
            mSpamConflict.add("u_spam");
            conflict.put("m_spam", mSpamConflict);
            HashSet uRealConflict = new HashSet();
            uRealConflict.add("m_spam"); uRealConflict.add("m_real");
            uRealConflict.add("u_spam");
            conflict.put("u_real", uRealConflict);
            HashSet uSpamConflict = new HashSet();
            uSpamConflict.add("m_real"); uSpamConflict.add("u_real");
            uSpamConflict.add("m_spam");
            conflict.put("u_spam", uSpamConflict);

            EventStructure eventStructure =
                        new EventStructure(events, conflict, causal);

            // Action
            MailerAction.initialise(eventStructure);
            Action receiveMail = Action.getActionByName("ReceiveMail");
            TrustDomain domain = receiveMail.getTrustDomain();

            // Trust policy initialise
            // We want some fixed values for certain principals
            TrustValue bottom = domain.getBottom();
            Action.Outcome secure = receiveMail.getOutcome("Valid");
            TrustValue.Triple triple = new TrustValue.Triple(10, 0, 0);
            TrustValue value = domain.valueOf(bottom, secure, triple);
            Logger log = Config.getConfig().getSmithLogger();
```

```
            TrustPolicy trustPolicy = new TrustPolicy(domain, log);

            // Evidence
            EvidencePolicy evidencePolicy = new EvidencePolicy();

            // Initialise Kernel
            secureKernel =
                        SecureKernel.initialise(
                        new AccessControlPolicy(domain, evidencePolicy),
                        trustPolicy, er, evidencePolicy, new Transport());
            trustPolicy.setTrust(cui, admin); // A trust reference
            trustPolicy.setTrust(vinny, value);
            trustPolicy.setTrust(waleed, value);
            trustPolicy.setTrust(jm, value);
            trustPolicy.setTrust(liz, value);
        } catch (Exception e) {
            System.err.println("Failed to start SECURE kernel.");
            e.printStackTrace();
        }
    }

    public static SecureKernel getSecure() {
        if ( setUp == null )
            setUp = new KernelSetUp();
        return setUp.secureKernel;
    }
}
```