THE TRUSTED FUNCTION IN SECURE DECENTRALIZED PROCESSING†

P. Tucker Withington

The MITRE Corporation
Bedford, Massachusetts 01730

The information processors in a decentralized computing system must trust each other enough to be mutually supportive, yet they must also protect themselves to maintain autonomy. In a decentralized system, data security is especially important because the effects of compromise or sabotage can be so wide-ranging.

The trusted function is an *ad hoc* solution to a problem with present data security models. This "consistency" problem, never previously addressed in a formal manner, is aggravated in the decentralized processing setting.

The paper examines the consistency problem and proposes an addition to existing security models to address the problem. Using the model, the impact of the trusted function on secure, decentralized-processing system design is assessed.

INTRODUCTION

The objective of this paper is to examine the concepts of computer security in the decentralized processing setting. One particularly weak point of present computer security systems, the "trusted function", becomes quite difficult in this new setting. The notion of the trusted function is scrutinized and a technique for supporting it in a decentralized processing system is discussed.

Background

The computer security problem has been thoroughly investigated for single-location systems. The present technological solution, the "security kernel", is simple enough and well-understood enough that many feel it can easily provide a solution for the distributed case. Some even believe decentralized processing will ease the security problem by providing each user with his own isolated hardware. It is not clear, however, that the security kernel or some of its "accessories" are so easily extended to the decentralized case.

Because of the new setting, a fresh examination of the security problem should be undertaken. In a decentralized system, the processors must establish some level of mutual trust to perform useful work (to maximize the advantage obtained by interconnection). Protection is even more of a consideration than in the single-location case, however, because the possibility of compromise or sabotage is enhanced by the same interconnection mechanism, and the payoff to an agent or saboteur is similarly raised.

Synopsis

This paper introduces the "consistency problem", a problem in current secure systems that is a major impediment to secure, decentralized processing. Current solutions are reviewed and some difficulties are discovered. After summarizing an exisiting formal security model, an extension to solve the consistency problem is proposed and its interpretation is discussed. Finally, the model is applied in the decentralized processing setting and the resulting implications examined.

THE CONSISTENCY PROBLEM

The purpose of decentralized processing[1] is to link together independent processors, each

†This research was supported by the Defense Advanced Research Projects Agency under ARPA Order No. AO-3338, Contract No. F19628-79-C-001, MITRE Project No. 8060. The views and conclusions contained in this paper are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

[1] The term "decentralized processing" is due to Karger [1].

CH1522-2C/80/0000-0067$00.75 © 1980 IEEE

providing useful services, in such a way as to enhance these services through their mutual support. To allow a user to take advantage of these interconnected processors, provision must be made for: common information storage, a uniform interface to processing tools, and a technique to control users on a network basis. [2,3]

From the security viewpoint, these three elements of decentralized processing correspond to: passive information containers, active information accessors, and the external system interface, respectively. Exactly how the notion of security affects these aspects of the distributed system is taken up after an examination of single-location security concepts.

Data Security

What are the requirements of data security? Most basic is the requirement that no unauthorized observation of protected data be allowed. This requirement has two facets. First, the protected data might be directly observed in an unauthorized manner — espionage; or second, the data might be indirectly or covertly observed with the aid of an inside accomplice, releasing information — treachery.

How do these requirements map into a computer system? One critical, but not obvious, point is that all objects in the system must be treated as data repositories — even the active observers and modifiers (they must have a place for the data they manipulate). With this simplification, we see that data security means: for any transfer of data from one repository to another (within the system), or to or from outside the system, it must be ensured that the transfer is "permitted".

The concept of permission is also known as the security policy. This policy determines what transfers of data are allowed. A straightforward policy, which can be shown to correspond to the security policy of the Department of Defense, can be defined as a function, mapping ordered pairs of containers into allowed or disallowed. The mapping defines whether the transfer of information from the first container of the pair is allowed to the second.

Because the permission function is independent of the information transferred, such a simple policy is known as an information flow model; it is the path the information flows along and not the information transferred that is regulated. While this does not exactly model the security system of the paper information domain, it has been accepted as a reasonable choice for the computer processing domain. A resulting problem, however, is how to best keep the two domains (internal-computer and external-paper) in correspondence, or consistent.

Consistency

The "consistency problem" involves establishing and maintaining protection information in the computer domain in agreement with the paper domain. As information enters and leaves the system, and while it resides in the system, the protection information about it must be kept consistent with the external environment.

Usually, this protection information is held in a "protection data base" that embodies the security permission function for all the information containers in the system. In an implementation, the access control mechanism enforces the security policy by referring to this protection data. A difficult point in early security modelling work was how this database would be maintained.

History

The "trusted function" evolved as a special purpose mechanism in response to the realization that the simplistic security model prevented one from accomplishing many tasks required in everyday use of the computer system (e.g., reclassifying old information, classifying new information). Initially, the trusted function was exempted from the model controls because it was trusted to violate the model only to keep the system consistent with the external environment.

Stork [4] presented a model for these trusted functions based on the "formulary concept": the idea that the correct classification of an information container could be determined from some computation on its contents. This concept was initially favored because it worked outside the existing security model, but in the end, opinion was that the formulary was impossible to establish[2]. Also, this model addressed only one facet of the consistency problem: downgrading. It did not address other operational requirements, as we shall see arise in a secure network design.

As implementations evolved, it became clear that there was a second type of trusted function that did not violate any security axioms, but was known to have an important security effect. These trusted functions dealt with the consistency problem by directly manipulating protection data. The mechanism was trusted to handle the protection information correctly.

Functions of that type have recently become known as "responsible software". These trusted functions did not fit either the security model or Stork's model, as they were not violating any model rule. Nonetheless, they were manipulating what was known to be highly sensitive data. As a result, an ad hoc[3] mechanism evolved [5,6] to provide the required function.

Mogilensky [7] provided the first recognition of these requirements as a general problem of interfacing distinct security systems. He

[2] Work continues on trying to maintain traces (colors) of the origin of information in a file such that a computation of its overall classification can be derived, but it is of an experimental nature.

[3] By ad hoc we imply: without formal modelling or specification.

attempted to define the requirements for communicating protection information from one domain to another. Schiller [8] finally succinctly defined the trusted function as providing "... the security related binding of computer system elements to the external environment".

In these early designs, a special interface to support the trusted function was implemented by a reserved key on the user terminal that caused activation of a trusted communication path. The trusted path provided two features not provided by the basic access control mechanism: a high degree of reliability in preventing the injection of spurious or misleading data[4], and the ability to accurately identify the participating parties in the communication.

Preview

What the trusted function and its associated interface attempted to do is to recognize the existence of a special kind of information in the computer system, different from the data of the system, and to provide a mechanism for handling it. Unfortunately, the full importance of this "control" information was never recognized in the single-location system work, and it was never successfully specified or modelled. Next, an attempt is made to do so.

TRUSTED FUNCTIONS

Present

The original security model dealt only with information and not information about information. Many attributes of protected information can be handled by the same mechanism that protects the information[5]. Some "meta" information does not fit into the model of security, however, because it is the model (or its representation). Still, it is an information system, and to deal with it correctly, a model should be developed. Figure 1 illustrates the existing information flow security model and highlights the meta-information of concern, the protection data base.

Initially, it is not clear that this meta-information has any analogue in the paper information domain. There, both the information distribution channel and the end user are trusted to maintain the protection class of information,

[4]Biba [9] has worked on a general policy to provide degrees of such protection to thwart sabotage; the trusted function was a first special case.

[5]These attributes are information in an equal or slightly higher protection class. The "directory" mechanism evolved as a way to handle this information, although it was eventually realized that the hierarchical organization of information was only a functional requirement and not security relevant. [10]

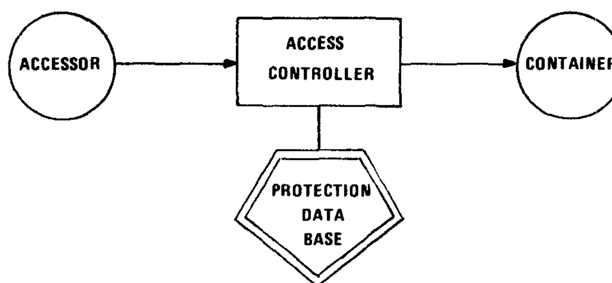

Figure 1. Existing Security Model

independent of its container. The binding of protection and information in the paper domain is not so obvious (or tenuous) as in a computer system.

Hypothesis

Let us consider the existing information flow model in figure 1. This model is primarily concerned with the flow of information to and from containers, within the system. Flow outside the system can be modelled as a container on the boundary of the system. The definition of permissible flow is held in the protection data base.

As we have noted, this protection data is not necessarily static. The protection data base could not be updated through the access controller, though, because its protection policy seemed different from that of the other data. The best determination of policy that can be arrived at by examining existing solutions is that the protection data base is protected on individual identity. It is the individual's privilege that allows him to examine or modify the protection data base.

To model the trusted function interface, we can use the same information accessing framework of the existing model, but with a new protection data base. This "trust data base" will associate an identity with the set of rights exercisable by that individual. These rights will specify what protection data can be accessed and in what way.

It also makes sense at this level to allow the trusted function interface to mediate access to its own trust data base. (The hierarchy of access mediators cannot go on forever.) Figure 2 illustrates the hypothesis and how it fits with the old model. This proposal is concisely defined below, by summarizing an existing information flow model and proposing an extension.

Information Flow Model

The information flow model to be reviewed models the problem of controlling the dissemination of classified information in a computer system. The approach is to control the access of the active agents in the system (who can cause information transfer) to the passive containers of information.

Setting. In the people/paper domain that is to be mimicked by the computer system, all "Data"

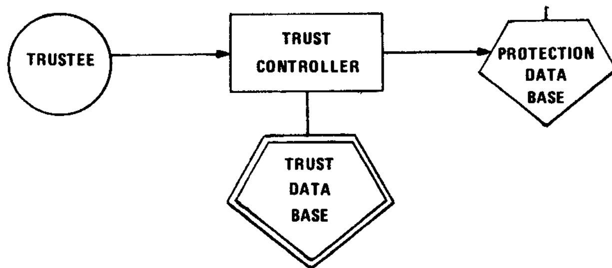

Figure 2. Proposed Model

is classified by an appropriate authority such that one can imagine the "Classification" of data as defining the set of observers allowed to view the data. The model does not examine this concept of classification; it takes the classification of the information in each of its containers as an external given.

Model. The model[6] is made up of **Repositories**, **Accessors**, **ProtectionClasses**, a **FlowRelation**, and a **ClassCombination** function.

where:

Repositories is a set of information containers; the containers define the parcels of information protected in the model. The value of a repository is the information it contains.

Accessors is a set of information manipulators, a subset of **Repositories**; the accessors are the model elements that can cause information flow. The information flow operation of an accessor consists of affecting the value of one repository based on a function of the values of other repositories.

The protection classes and flow relation make up the protection data base. The protection data base is the model representation of the information control policy of the paper domain. It is defined as follows:

ProtectionClasses is the partitioning of **Repositories** into classes according to the sensitivity of the information they contain (or can transfer). For two **repositories** to be in the same **protectionClass**, the "Classification" of their data must be equal.

FlowRelation is a partial ordering of the **ProtectionClasses** reflecting the relationship of the sets of people allowed to access each class. If **oneProtectionClass** is higher than **anotherProtectionClass** in the **FlowRelation**, then the "Classification" of repositories in **oneProtectionClass** is more restrictive than that of **anotherProtectionClass**. (Here, "more restrictive" means only a proper subset of people can view it, so in a sense its classification is "higher".)

Finally,

ClassCombination is a function on a pair of input protection classes that defines the class of the result of any function on a pair of values, one from each input class. The resulting **protectionClass** is the class that can be observed only by the set of observers allowed to view both the input **protectionClasses** under "Classification".[7]

In the paper domain, we think of security as meaning no information ever gets to a place where someone who isn't allowed to see it can see it. Under the model, this idea can be stated simply: the model system is secure if and only if no sequence of operations can cause information to flow from its container to a container lower in the flow relation (thus incorrectly increasing the set of observers of that information).

Security. We can state the idea of security as an axiom about individual (indivisible) operations. (Preserving this axiom for single operations can be shown to imply the security of aggregate operations.)

Security Axiom — Assuming a secure initial condition, if an **accessor** can cause information to flow from a set of **sourceRepositories** to a **destinationRepository** only if the **ClassCombination** of the **protectionClasses** of the **sourceRepositories** is not higher than the **protectionClass** of the **destinationRepository** under the **FlowRelation**, the system will remain secure.

Tranquility. In early use of the information flow model, an additional principle [12] was assumed.

Tranquility Principle — The **protectionClass** of an active (accessible) **repository** will not change during normal operation.

The tranquility principle is in effect a statement that the model is an in vitro experiment. The additional principle is an assumption in the model that must be shown for a "real" system.

Problem. The tranquility principle indicates that the model assumes that the paper concept of "Classification" is static. The model concept of classification is static in that a repository does not move from one protection class to another.

[6]The model described is a review of a detailed state-transition model that has been developed by Denning [11].

[7]In her paper, Denning showed that **ProtectionClasses**, together with **FlowRelation** and **ClassCombination**, forms a universally bounded lattice.

This assumption is inappropriate because it inhibits frequently required operations. As discussed, this problem is presently avoided by allowing violations of the security axiom. A preferred alternative would be to modify the model to account for the reclassification requirement. (As also mentioned, Stork's proposal has not been pursued because of the difficulty in specifying the formulary.)

Trust Model

We now propose an alternative model, based on the *ad hoc* mechanisms that have evolved in secure system implementations. This model will provide a basis for discussion for the remainder of the paper.

Setting. In the paper domain, we speak of reclassifying "Data". Since the model protects repositories, the operation of reclassifying is best modelled as a repository changing its protection class. Since the model does not embody the classifying authority, the reclassification *decision* must come from outside the model. This decision can be thought of as based on the people/paper concept of "Custodianship", where the custodian of any data is responsible for its classification being maintained correctly.

Model. The extended information flow model to handle reclassification (and thus trusted functions), will be called the Trust Model. It is concerned with the previous **Repositories** and **ProtectionClasses**, and replaces the tranquility principle by adding **Trustees** and a **TrustMap** relation,

where:

> **Trustees** is a set of agent identifiers of those allowed to manipulate the protection data base. They are *trusted* to maintain security by placing **repositories** in the appropriate **protectionClass**.
>
> **TrustMap** is a relation of **Trustees** and **Repositories** defining which trustee is allowed to maintain the classification of each repository. If a **trustee** is related to a **repository** by **TrustMap**, he has "Custodianship" of the data in that **repository**. (The relation may be a function whose value determines the specific privileges and responsibilities of the custodian.)

In the paper domain, the custodian of data is trusted to maintain its appropriate classification by not releasing it improperly and by upgrading it or downgrading it as required. Careful selection of custodians and significant legal penalties make the correct discharge of this responsibility highly probable.

Trustworthiness. Under the proposal, the trust mechanism allows paper domain custodians to perform their duties with respect to model information. The mechanism described is intended to enforce the requirement for responsibility through an additional axiom.

> *Trust Axiom* — *If* a **trustee** moves a **repository** from **oneProtectionClass** to **anotherProtectionClass**, *only if* that **trustee** is a custodian of the **repository** according to the **TrustMap**, then the reclassification can be trusted to leave the system secure.

The claim is that enforcing this axiom when allowing access to the protection data base of a computer security system will solve the consistency problem in a secure fashion.

Properties

The appendix introduces an example specification to motivate some important properties that must be enforced for a proper implementation of the trust model. Here, the properties are quickly reviewed.

Authority. Each paper domain custodian must be correctly associated with a representing trustee, for proper determination of rights. A password-like mechanism is a possible solution.

Continuity. Trusted changes to the protection data of an object must ensure that all current accesses to the object are still allowed. This check is required to prevent breaches of security from a potentially inconsistent state. Forcing recomputation of all existing accesses to the object is a possible solution.

Believability. Each trusted request must be conveyed from the paper domain to the computer domain in an integritous fashion, to prevent spoofing. A protected communication channel is a possible solution.

Summary

In addition to the trust axiom, three properties must also be enforced in an implementation to solve the consistency problem correctly. How these four requirements affect the design of a decentralized processing system is taken up in the next section.

DECENTRALIZED PROCESSING

How are the elements of a decentralized system affected by security? What is the correct interpretation of our new model in a decentralized case? Ignoring the trusted function for the moment, the problem is how to create the effect of access control in a system where the information containers, information accessors, the access controller(s), and the protection data base(s) are not necessarily centrally controlled.

Requirements

First, a uniform policy must be established. Even if initially different policies are enforced by each site, for the sites to communicate, the policies must map into each other; thus establishing a uniform policy. To enforce the policy, some

access control mechanism must be placed between all containers and accessors. Two obvious choices are to control all the accessors or to control all the containers[8].

Single-location secure systems provide a mechanism for protecting all containers; but, must each access controller know about every accessor-container combination in a network; or even all combinations with the objects he controls? Fortunately not, otherwise an information explosion would occur. Since the model establishes object classes that partition all objects according to their protection requirements, each access controller need only know what classes its objects fall into and understand how the security policy relates classes to one another.

Finally, a mechanism must be established for communicating a protection class along with information when it moves from the domain of one access controller to another. One way to communicate protection classes is to have separate protected network channels for each class of information. This arrangement works well with existing single-location systems; they can deal with communication channels simply as a new type of external object.

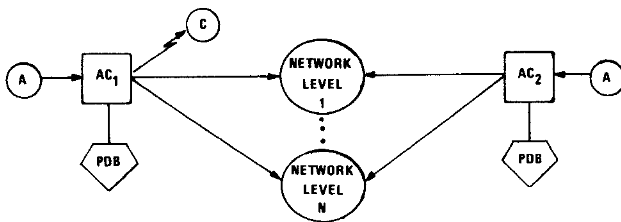

Figure 3. Existing Network Design

Figure 3 gives a representation of the design presently being pursued in an existing network. [1,13] Here, the hope is to connect together existing single-location systems in a fashion similar to the Arpanet, only with careful attention to security. Each information channel supplied by the network is treated as an object by both access controllers. While this design supports remote information accessing, it does not support all the functions desired in a network operating system.

<u>Special Functions</u>. If we examine the elements of a decentralized processing system, we discover additional needs. Remote reclassification, remote use of tools, and remote login all present additional requirements.

A protected channel per class works well for transmitting access data along with information. Information moving from one location to another may carry its protection class in this way. The ability to change the class of information, however, is not supported by this mechanism.

For the user to invoke tools on a remote host (as if he were local) his network connection should support the same facilities as his terminal. In particular, there is a need for the user to change his working classification at various times throughout a session, regardless of his maximum or login classification.

Finally, the user interface to the distributed system must be provided remotely. If a user is to be authenticated (logged in) remotely from his authentication data base, or if his authentication is to be propagated to another node in the system, the network data channels do not suffice.

All these requirements can be reduced to the requirement of providing a trust interface in the distributed case, over the network. Assuming that a trust interface (enforcing the trust axiom and obeying the continuity requirement) exists for the single-location security systems in the network, we discuss below how the authority and believability requirements affect the design of a secure network to interconnect them.

<u>Network Security</u>

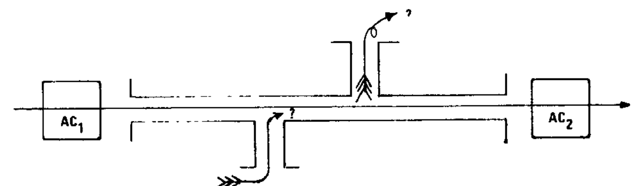

Figure 4. Secure Network Connection

Figure 4 shows a diagram of a secure network connection between two single-location machines as it might appear in the network of figure 3. In this system, the data channels are provided by a link-encrypted packet switch network, where the switches are single-location, kernelized minicomputers. The figure illustrates why data channels cannot support trusted communication.

Because existing security models only restrict data flow to a particular class, and, because of the nature of a class, the source and destination of a data channel are not uniquely defined (they might be any member of the class). The authority principle can thus be subverted as in figure 4: although the questionable data flows (arrows) are to the correct level, under the security policy we

[8] A less obvious choice is to control all communication paths, an option as yet unexplored in securing single-location machines, but enticing in networks, where the communication paths are explicit.

have no guarantee of the individual identity of the sender or receiver.

In addition, a plain security policy (in particular, one that does not include some form of integrity, c.f., [9]) does not provide the additional believability principle of our model. If the policy cannot distinguish between an accessor reading information from a lower class and the symmetric but separate operation of an accessor writing information to a higher class, the second operation can be used to sabotage believability (lower arrow in figure 4).

What is needed is a technique to authenticate the parties in the communication (for authorization) and to provide additional protection (for believability) to meet the new trust requirements.

The communications network in a secure system has been modelled as an data object shared by two (or more) access controllers. A trusted communication channel to support a secure virtual connection could be provided by a similar shared object for trust controllers. Figure 5 gives a conceptual picture of what is desired — a parallel but separate network, enforcing the trust model requirements.

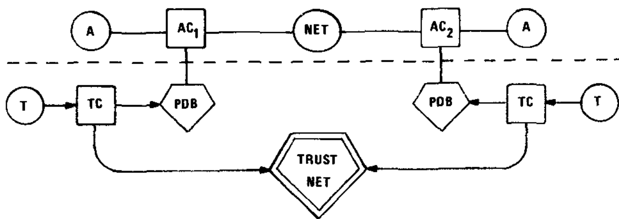

Figure 5. Example Trust Network

Solution. The requirement for authentication in communication is usually achieved through a recognition code or password. A password is only useful if a limited number know it, and if it can be communicated without being overheard.

Assuming we cannot create a perfect channel, the protection requirement for integrity could be achieved through redundancy. This redundancy must be such that any attempt to modify the data could be detected, and perhaps corrected.

Cryptography can help provide both these features. By disguising the information content of a message, enciphering a message makes it very difficult to change the cipher version in a way that is meaningful when deciphered. Enciphering data raises the inter-bit dependency of the data sufficiently that a small amount of redundancy in the unenciphered data will allow detection of modification. [14,15]

Also, knowledge of a cipher key can act as a password. The source of decipherable code is limited to those who possess the key. By design, enciphered text does not reveal the encoding key, so the password cannot be compromised. By limiting key distribution, the authentication can be made as specific as necessary. [14]

Implementation. In an implementation, a trusted function interfaces to the access monitor through privileged calls. The trusted communication channel could be implemented by such a call.

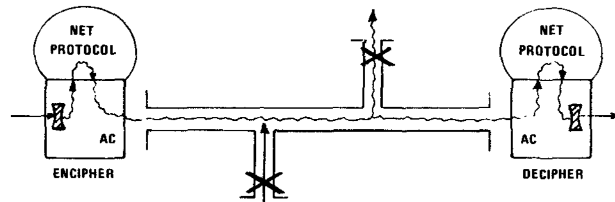

Figure 6. Possible Trust Net Implementation

Figure 6 shows a minimal implementation. The proposal is to provide privileged calls for enciphering and deciphering trust information. These calls will add and make appropriate redundancy checks to provide integrity. These calls will also add identification information to each trust message to provide authentication. Through encryption, these calls provide the trust network as a specially protected virtual channel over the existing data network.

Because of their importance, the privileged calls must be specified and implemented as part of the access controller and scrutinized for the same level of credibility. The intent is that the operation of these two calls can be verified at a significantly lower expense than trying to verify the network end-to-end protocol module or a whole new channel.

Kent [16] has discussed at some length the choice of a cipher scheme to allow authentication and detection of modification. While these mechanisms are fairly expensive, the design proposed here would limit their use to protecting trust data and thus make their cost reasonable.

Once the trust information is enciphered by the privileged kernel call, it becomes an unclassified object (its information content is zero to a non-key holder). The enciphered package can then be delivered using existing network commands.

Impact. The end-to-end encryption channel provided by the proposed privileged calls essentially provides an "out-of-band" signal that can be used by the trusted functions to communicate trust information over a secure data network.

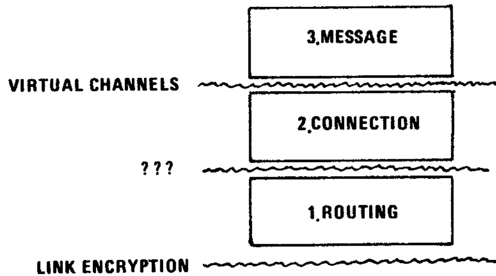

Figure 7. Network Protocol Levels

Figure 7 illustrates how the proposal relates to existing secure network protocols:

The bottom level (1) represents the existing protection provided by secure networks. Here, the internet protocol, the host addresses, and other message data are unenciphered when in a network node. Protection is provided during transmission between network nodes by link encryption. The dangers of this protection alone are known, and work is under way to solve the problems in several data networks.

The second level (2) represents the end-to-end connection protocol that attempts to address the denial of service problem in existing protocol designs [17]. Attempts to protect connections at this level have been unsuccessful to date in that they exhibit many of the same problems as the bottom level [18], and yet, are significantly more difficult to implement.

The top level (3) represents the present proposal, a message level protocol. The intent is to use a single connection (provided by the next lower level) to support at least two virtual channels by appropriately distinguishing messages.

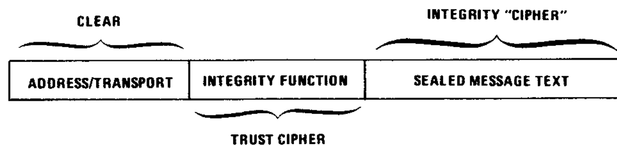

Figure 8. Sample Trust Message Content

Figure 8 shows a possible construction of a trust message and how it fits in the lower network protocols. Only a small amount of information about the message need be protected using the network "trust channel" cipher: an integrity function to provide authority and believability for the rest of the message. (This function may be a secondary cipher — different for each message.) The integrity information acts as the "seal" of the sender. If the receiver finds the seal unbroken, he knows he can trust the contents.

Because the amount of information transmitted using the trust cipher is small, the trust cipher scheme need not be fast (it can be implemented in software). Similarly, it need not be re-keyed often.

If the lower level protocols can protect the connection from a security standpoint (at the highest level of data to be transmitted), the trust cipher need not protect against espionage from outside the system. Its duty is simply to inhibit the effect of sabotage (or error) within the system.

Summary

An "out-of-band" channel has been proposed as one way of providing the properties required for trusted function implementation in a decentralized system supported by a secure data network. Encryption was proposed as a technique for providing that out-of-band channel, but all the power of traditional encryption schemes is not required. Perhaps a better mechanism can be found.

CONCLUSION

The computer security problem (the need to correctly regulate the sharing of data in a computer system) has been thoroughly investigated for single-location systems. The present technological solution to the problem is the **security kernel** mechanism. This simple mechanism assigns a protection class to each identifiable information repository and then regulates the flow of information to be only between repositories of compatible classes.

A deficiency discovered in the security kernel is how to correctly assign a protection class to information from outside the system (e.g., information entered at a terminal) and how to maintain consistency between the paper information domain and the computer information domain (e.g., to follow reclassifications). In the single-location system, these problems have been solved by an *ad hoc* mechanism called the **trusted function.** This function provides a reliable interface to the protection database of the kernel to allow updates to that data.

In the decentralized processing case, where information repositories are scattered and the protection data base is similarly distributed, the trusted function is even more important because the consistency problem is exacerbated. The operation of the trusted function has been reevaluated in this new setting.

The problem of how existing networks can accommodate secure information sharing is seen to hinge on support of the trusted function operation in a more general case. Determining the

requirements of the trusted function leads to formalizing the concept of communicating protection information between security domains and the development of a **Trust Model**.

The model distinguishes the protection information as a special type of data that must be communicated in a trustable manner since it is crucial to the general protection mechanism. Any solution to this problem will impact the design of a decentralized processing network and its protocols.

Finally, software-supported, end-to-end **encryption** has been proposed as a possible method for providing the requirements of trusted communications. The impact of this mechanism on protocols is seen to be minimal. Nevertheless, further investigation is required to determine the best possible solution.

ACKNOWLEDGEMENT

The author acknowledges the motivation and help of S. R. Ames, Jr., Dr. J. G. Keeton-Williams, and Dr. J. K. Millen of the MITRE Corporation in developing the proposed model.

EXAMPLE

An example multiplexed processing system specification is given to point out some important properties of the trust model that must be considered in an implementation.

Trust System

Figures 9-17, give the elements of the trust system. Figure 9 gives the top level module interface that defines the behavior of the system. The system is made up of a scheduler, a program processor, a kernel, and a trusted interface. It knows of a number of requests, their associated parameters, and can return a number of answers. Its operation is defined by the "Schedule" function. "InputBuffer" and "OutputBuffer", defined in the kernel, are the interfaces to the system.

Figure 10 gives the scheduler module. This model defines the overall operation of the system, which consists of eternally choosing a process and doing a computation for it. The variable "currentProcess" defines the multiplexing of the system.

Figure 11 is the program processor, responsible for computations. The "Compute" function defines the basic processor operation, which is to open the program file, retrieve a request and parameter for execution, perform the operation, place the result in the accumulator, and increment the program counter to point to the next portion of the program.

The function "ArithLogic" defines the non-security-related requests that can be computed by the system. It supports binary functions on the accumulator and the current readable memory portion.

Figure 12 shows the kernel interface, which defines all the security-related requests that can be computed by the system. The kernel supports storage, processes, and input/output in a secure fashion using the protection database.

Figure 13 gives the protection data module, which defines the permissible information flows. The function "PermissibleFlow" decides whether information may flow in a specified direction between a memory object and the current process. The function bases its decision upon the "Class" database, which for the moment is assumed correct.

Figure 14 is the storage portion of the kernel. It ensures that information motion requests by the current process to and from the memory are carried out in accordance with the protection policy. Storage supports three virtual memory areas for each process: one each for reading input, writing output, and accessing the program.

The "Map" function checks access to a specified area for a specified purpose and records the result. The functions "Get", "Put", and "Execute" are used to access the area mapped for input, output, and program, respectively.

Figure 15 shows the processes provided by the kernel. The process module keeps track of the program counter and accumulator registers for each process and ensures that an executing process can access only its own registers.

Figure 16 gives the input/output module of the kernel. This module ensures that information motion requests to and from the interface buffers obey the protection policy. The interface buffers are the areas of memory where devices that interface to the paper domain exchange information with the system. The functions "Input" and "Output" are used to access the memory area appropriate to the I/O device or "port" desired.

Figure 17 defines the trusted interface to the system. It is an attempt to address the earlier assumption of correctness regarding the "Class" protection database. "ChangeClassOf" provides a function to change the class of storage objects in the system. Unfortunately, it depends on another mysterious database, called "Proprietorship".

Observations

Two observations will be made about the system before analyzing the trust interface:

First, note that "Map" in the storage module obviates the need to compute "PermissibleFlow" each time the memory is to be accessed. This organization has been found expedient in practice because of the high frequency of memory operations, but it leads to some complications in enforcing the proposed model, as will be seen below.

Second, the interface buffers in the input/output module illustrate the implementation issues we wish to address. These buffers are storage objects classified by the "Class" function in the protection data module, but the system has no way of assuring that external access to the devices they interface is consistent with the internal assignment. This problem is one facet of the consistency problem.

Analysis

The trusted interface in figure 17 is the implementation of the extended model to address the consistency problem. The function purports to allow a trusted user to update Class to keep it consistent with the paper domain. Three additional properties must be assured to achieve this goal in an implementation, however; in addition to enforcing the trust axiom we require: "authority", "continuity", and "believability".

Authority. Some mechanism is required to ensure that only the correct authority is permitted to exercise the "ChangeClassOf" function. In the example specification, this assurance is indicated by the argument "user". We have no reason to believe, however, that a malicious user will properly identify himself.

The problem is similar to the problem of knowing the class of the input/output ports. For the ports, however, we have some belief about the location of the devices being static and protected physically. These facts allow the "Class" function to be preset for ports.

A similar mechanism is needed to identify the user on an individual basis, but we have no reason to believe a user's location static. The currently accepted mechanism for authenticating the user of an interface is the password mechanism[9].

Continuity. A scenario of operations that can occur in the system is for one user to open a storage object for writing, and then for another user to lower the class of that object. Under the present implementation specification, this scenario leaves an information path for the unwitting first user to leak information. Even though the downgrade was "trusted", something went wrong.

There are two immediate remedies to this problem: The memory unit can check the class of each object at each access; a solution experience tells us is infeasible. Or, the trust function may not change the class of an object in use, a procedure reminiscent of the old tranquility principle. A third option would be for the trust function to ensure that all users of the object can still use it or otherwise disconnect (unmap) them, thus maintaining the continuity of the security policy.

Believability. Since the request buffers are simply storage objects, protected only by class, requests entered by devices using these buffers may be sabotaged by anyone of appropriate classification. Because the authority requirement depends on the user's identity, some means is needed to provide a believable request channel that can preserve the integrity of the user's request, also based on identity (rather than just classification).

Solution

Figure 18 shows an updated trusted interface that attempts to remedy the three implementation problems discussed above.

A new set of interface channels, "TrustChannel", is added to provide a trustworthy and dependable communication with the user for "Believability". While these channels are still associated with the existing ports to devices, they do not allow general access. Basically one can think of them being input buffers under the control of the kernel or trusted process. (If the trustworthiness of the normal device is of concern, a special device can be interfaced to the trusted channel.)

The "ChangeClassOf" function is modified to read only from a trusted channel. The user's purported identity is read from the trust channel and will be verified against a password. (The protected channel is used here to maintain the secrecy of the password, actually an unintended use of the channel.) The arguments for class change, object and new level, are also read from the trusted channel. If the password passes and the user is the proprietor of the specified object, then the main body of the function is entered.

An additional check is now made to see that the object is not "inuse". This check represents the second of the "Continuity" options. (The third was not pursued only because the existing permission function did not allow easy check of access other than by the current user.)

If the object is not active, its new level is set, and the proprietor is notified by the trust channel. (Consistency would not be fulfilled if the proprietor could be spoofed about the result.)

Summary

An example implementation of the trust axiom as a trusted interface for a multiplexed processing system has been presented. We have also noted three properties that must be enforced in an implementation to solve the consistency problem correctly: authority, continuity, and believability.

---

[9] Other, more-sophisticated, authentication mechanisms are under development and should be used when feasible.

```
module SystemInterface ≡
begin
 exports InputBuffer, OutputBuffer,
        with RequestName, ParameterType, AnswerType;

 includes Scheduler;
          ProgramProcessor,
          Kernel,
          Trusted;

 RequestName ≡ (Map, Get, Put, Registers, Input, Output,
               ArithLogic, ChangeClassOf);
 ParameterType ≡ setof (StorageChunk, Direction,
                        ProcessNumber, RegisterName,
                        PortName, BinaryFunction,
                        ProtectionClass, UserName);
 AnswerType ≡ oneof (Can't, Ok, Value);

 Schedule;

end SystemInterface;
```

Figure 9. Trust System Interface

```
module Scheduler ≡
begin
 exports Schedule;

 exports currentProcess to Storage, Processes, Protection;

 function Schedule() ≡
 {
  for(ever)
  {
   currentProcess ← Choice(ProcessNumber, currentProcess);

   Compute;

   continue;
  }
 }

end Scheduler;
```

Figure 10. Scheduler Module

```
module ProgramProcessor ≡
begin
 exports Compute, ArithLogic;

 function Compute() ≡
 {
  request:RequestName,
  parameter:ParameterType;

  Map(Registers(programCounter), program);
  request ← Execute.request;
  parameter ← Execute.parameter;

  Registers(accumulator) ← request(parameter);

  Registers(programCounter) ←
   Next(StorageChunk, Registers(programCounter));
 }

 function ArithLogic(function:BinaryFunction)value:Value ≡
 {
  value ← function(Registers(accumulator), Get);
 }

end ProgramProcessor;
```

Figure 11. Program Processor Module

```
module Kernel ≡
begin
 exports Map, Get, Put,
         with StorageChunk, Direction;
         Registers,
         with ProcessNumber, RegisterName;
         Input, Output,
         with PortName;

 includes ProtectionData,
          Storage,
          Processes,
          InputOutput;

end Kernel;
```

Figure 12. Kernel Module

```
module ProtectionData ≡
begin
 exports PermissibleFlow,
        with ProtectionClass, Permission;

 exports Class to Trusted;

 function
  PermissibleFlow(location:StorageChunk,
                  direction:Direction)permission:Permission ≡
 {
  permission ← if(direction = input | direction = program)
               then if(Class(location) ≤ Class(currentProcess))
                    then allowed;
               else if(direction = output)
               then if(Class(location) ≥ Class(currentProcess))
                    then allowed;
               else denied;
 }

 Class : array(StorageChunk U ProcessNumber)
         of ProtectionClass initially "secure";
 ProtectionClass ≡ (anyoneCanSeeIt,
                    kindOfImportant,
                    realSensitive);
 Permission ≡ (allowed, denied);

end ProtectionData;
```

Figure 13. Protection Data Module

```
module Storage ≡
begin
 exports Map, Get, Put,
        with StorageChunk, Direction;

 exports Execute to ProgramProcessor;

 function Map(direction:Direction, which:StorageChunk) ≡
 {
  if(PermissibleFlow(which, direction) = allowed)
  then VirtualMemory(currentProcess, direction) ← which;
 }

 function Get()value:Value ≡
 {
  if(VirtualMemory(currentProcess, input) ≠ null)
  then value ←
       Memory(VirtualMemory(currentProcess, input));
 }

 function Put() ≡
 {
  if(VirtualMemory(currentProcess, output) ≠ null)
  then Memory(VirtualMemory(currentProcess, output)) ←
       Registers(accumulator);
 }

 function Execute()value:Value ≡
 {
  if(VirtualMemory(currentProcess, program) ≠ null)
  then value ←
       Memory(VirtualMemory(currentProcess, program));
 }

 Memory : array(StorageChunk) of Value
          initially Memory(firstChunk) ≡ "boot program";
 VirtualMemory : array(ProcessNumber, Direction) of StorageChunk;
 StorageChunk ≡ (firstChunk, secondChunk, ... lastChunk);
 Direction ≡ (input, program, output);

end Storage;
```

Figure 14. Storage Module

```
module Processes ≡
begin
 exports Registers,
        with ProcessNumber, RegisterName;

 Registers ≡ AllRegisters(currentProcess);

 AllRegisters : array(ProcessNumber, RegisterName) of Value
                initially AllRegisters(firstProcess,
                                       programCounter) ≡ firstChunk;
 ProcessNumber ≡ (firstProcess, secondProcess, ... lastProcess);
 RegisterName ≡ (accumulator, programCounter);

end Processes;
```

Figure 15. Process Module

```
module InputOutput ≡
begin
 exports Input, Output,
        with PortName;

 exports InputBuffer, OutputBuffer to SystemInterface;

 function Input(port: PortName)value:Value ≡
 {
  if(PermissibleFlow(InputBuffer(port), input) = allowed)
  then value ← Memory(InputBuffer(port));
 }

 function Output(port:PortName) ≡
 {
  if(PermissibleFlow(OutputBuffer(port), output) = allowed)
  then Memory(OutputBuffer(port)) ←
        Registers(accumulator);
 }

 interface InputBuffer : array(PortName) of StorageChunk;
 interface OutputBuffer : array(PortName) of StorageChunk;
 PortName ≡ (frontPanel,
             consoleTerminal,
             anotherTerminal,
             bigTape,
             linePrinter,
             aNetConnection);

end InputOutput;
```

Figure 16. Input/Output Module

```
module Trusted ≡
begin
 exports ChangeClassOf,
        with UserName;

 function ChangeClassOf(user:UserName,
                        which:StorageChunk,
                        new:ProtectionClass) ≡
 {
  if(user = Proprietorship(which))
  then Class(which) ← new;
 }

 Proprietorship : array(StorageChunk) of UserName
                  initially "trusted";
 UserName ≡ (me, you, ... others);

end Trust;
```

Figure 17. Trusted Interface Module

```
module Trusted ≡
begin
 exports ChangeClassOf,
        with UserName;

 exports TrustChannel to SystemInterface;

 function ChangeClassOf(argumentChannel:PortName) ≡
 {
  user:UserName,
  password:Value;
  which:StorageChunk,
  new:ProtectionClass;

  user ← TrustChannel(argumentChannel).user;
  password ← TrustChannel(argumentChannel).password;
  which ← TrustChannel(argumentChannel).storageChunk;
  new ← TrustChannel(argumentChannel).protectionClass;

  if(password = Password(user))
  then if(user = Proprietorship(which))
       then if(∃ process ∈ ProcessNumber, direction ∈ Direction $
                 VirtualMemory(process, direction) = which)
            then TrustChannel(argumentChannel) ← Can't;
            else
            {
             Class(which) ← new;
             TrustChannel(argumentChannel) ← Ok;
            }
 }

 Proprietorship : array(StorageChunk) of UserName
                  initially "trusted";
 interface TrustChannel : array(PortName) of Value;
 Password : array(UserName) of Value initially "trusted";
 UserName ≡ (me, you, ... others);

end Trust;
```

Figure 18. Better Trusted Interface

REFERENCES

1. P. A. Karger, "Non-discretionary Access Control for Decentralized Systems," LCS/TR-179, MIT Laboratory for Computer Science, Cambridge, Massachusetts, May 1977.

2. R. H. Thomas, R. E. Schantz, and H. C. Forsdick, "Network Operating Systems," RADC-TR-78-117, Rome Air Development Center, Griffis Air Force Base, New York, May 1978.

3. D. P. Geller and K. Sattley, "National Software Works User's Reference Manual System Version 2.1," CADD-710-2611, COMPASS, Wakefield, Massachusetts, October 1977.

4. D. F. Stork, "Downgrading in a Secure Multilevel Computer System: The Formulary Concept," ESD-TR-75-62, The MITRE Corporation, Bedford, Massachusetts, June 1974.

5. D. E. Bell, R. S. Fiske, M. Gasser, and P. S. Tasker, "Secure On-line Processing Technology — Final Report," ESD-TR-74-186, The MITRE Corporation, Bedford, Massachusetts, August 1974.

6. W. L. Schiller, "The Design and Specification of a Security Kernel for the PDP-11/45," ESD-TR-75-69, The MITRE Corporation, Bedford, Massachusetts, March 1975 (AD A01171).

7. J. Mogilensky, "A General Security Marking Policy for Classified Computer Input/Output Material," ESD-TR-77-259, The MITRE Corporation, Bedford, Massachusetts, May 1975 (AD A016467).

8. W. L. Schiller, "The Design and Abstract Specification of a Multics Security Kernel," ESD-TR-77-259, volume 1, The MITRE Corporation, Bedford, Massachusetts, January 1977.

9. K. J. Biba, "Integrity Considerations for Secure Computer Systems," ESD-TR-76-372, The MITRE Corporation, Bedford, Massachusetts, April 1977.

10. S. R. Ames, Jr., "File Attributes and Their Relationship to Computer Security," ESD-TR-74-191, Case Western Reserve University, Cleveland, Ohio, June 1974.

11. D. E. Denning, "A Lattice Model of Secure Information Flow," Communications of the ACM, volume 19, number 5, May 1976, pp. 236-243.

12. D. E. Bell and L. J. LaPadula, "Secure Computer Systems," ESD-TR-73-278, volumes 1-3, The MITRE Corporation, Bedford, Massachusetts, April 1975.

13. G. D. Cole and D. K. Branstadt (editor), "Design Alternatives for Computer Network Security," publication 500-21, volume 1, National Bureau of Standards, Washington, District of Columbia, January 1978 (PB 276 771).

14. G. J. Popek, and C. S. Kline, "Design Issues for Secure Computer Networks," Operating Systems — An Advanced Course, R. Brayer et al., (editors), Springer-Verlag, Berlin, Germany, 1978, pp. 518-546.

15. R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," Communications of the ACM, volume 21 number 12, December 1978, pp. 993-999.

16. S. T. Kent, "Protocol Design Considerations for Network Security," NATO Advanced Studies Institute on Interlinking of Computer Networks, Bonas, France, August 28-September 8, 1978.

17. J. Postel (editor), "Transmission Control Protocol (TCP) — Version 4," IEN: 81, Information Sciences Institute, Marina del Rey, California, February 1979 (AD A067072).

18. M. A. Padlipsky, D. W. Snow, and P. A. Karger, "Limitations of End-to-End Encryption in Secure Computer Networks," ESD-TR-78-158, volume 1, The MITRE Corporation, Bedford, Massachusetts, May 1978.