UC Irvine
ICS Technical Reports

Title
A protection model and its implementation in a data flow system

Permalink
https://escholarship.org/uc/item/65m4c9kk

Author
Bic, Lubomir

Publication Date
1980-01-15

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

A PROTECTION MODEL AND ITS IMPLEMENTATION IN A DATAFLOW SYSTEM

by

Lubomir Bic

Technical Report # 148

January 15, 1980

Keywords and Phrases: protection, keys, interprocess communication, proprietary services, selective confinement problem, Trojan Horse problem, dataflow.

CR Categories: 4.35

This work was supported by NSF grant MCS-76-12460: The UCI Dataflow Architecture Project.

ABSTRACT

A protection model is presented for a general-purpose computing system based on keys attached as 'seals' and 'signatures' to values exchanged among processes. A key attached to a value as a 'seal' does not prevent that value from being propagated to any place within the system; rather, it guarantees that the value and any information derived from it cannot leave the system unless the same key is presented. A key attached to a value as a 'signature' is used by a process to verify the origin of the received data. Solutions to problems from the areas of interprocess communication and proprietary services are given.

CONTENTS

1. Introduction and Objectives

2. An Intuitive Description of the Model

3. Implementation of the Model in a Dataflow System

3.1. Basic Dataflow Principles

3.2. Dataflow Processes

3.3. Implementation of Protection Mechanisms

4. Application of the Protection System

4.1. Proprietary Services

4.2. The Selective Confinement Problem

4.3. Private Interprocess Communication

5. Conclusion

1. Introduction and Objectives.

In recent years the need for better and less restrictive protection mechanisms has emerged but the drawback most common in advanced protection systems is their complexity in both use and understanding. In addition, meny well known protection problems still have no satisfactory solution in those systems. The goal of this paper is to present a protection mechanism that is easily understood by the (user) programmer, yet powerful enough to allow the solution of a large variety of protection problems. This mechanism is defined by a very small set of primitive operations that may be incorporated as part of a high-level language. Thus the implementation and enforcement of protection policies does not require that the user leave the domain of the language in which his programs are written as is the case in most contemporary systems (no switching to a job or file control language is necessary). This also means that operating systems and user programs both utilize the same mechanisms to implement their own protection policies.

In the sequel, we give a description and a possible implementation of the proposed mechanism in the context of a dataflow system. We will demonstrate that despite the conceptual simplicity of the system we are able to give satisfactory solutions to well known protection problems, for example from the areas of private interprocess communication and proprietary services, in particular the 'Selective Confinement Problem' /Lam73/ and the 'Trojan Horse Problem' /Sch72/.

We consider a computing system to be a collection of independent processes each of which is in possession of a number of objects, where an object may be any piece of data. (In the sequel we will use the terms 'object' and 'value' as synonyms since in dataflow any object is treated as a value). Processes communicate by sending messages, where a message is a copy of some value; that is, no sharing of objects among processes is allowed.*

Our goal is to provide mechanisms which can be employed by the programmer to satisfy the following two basic requirements:

a) Each process sending some value v to another process is able to 'seal' the value such that only the intended destinee can 'unseal' and thus utilize v.

b) The sender of v is able to uniquely 'sign' v thus allowing the receiver to authenticate the sender.

Both of the above requirements should hold even if the value is being passed via other 'courier' processes. If this is the case then any two processes are able to exchange secret or private data without the risk that a third process could utilize that data (due to requirement a) or substitute it by a 'fake' (due to requirement b). We argue that the use of a proprietary service may also be viewed (and implemented) as interprocess communication: If the user and the service are two separate processes then the user is sending a package of information

---

*In dataflow these requirements are always satisfied. Some operating systems also provide an equivalent view of inter-process communication, in which case the results of this paper are applicable.

(e.g. the arguments) to the service which then produces a new package of information (the results) that is then sent back to the user. Thus the user of a service is considered as both the sender and the receiver of an information package which is being passed through and modified by an intermediate process - the service. In this way the service is similar to a 'courier' process, the only difference being that the service modifies the data before forwarding it to the receiver. With this point of view we will focus on providing mechanisms for interprocess communication according to the requirements a) and b) above, and demonstrate that these requirements are also sufficient to satisfy the needs of proprietary services.

2. An Intuitive Description of the Model.

We introduce a new data type called key. Each key k consists of a value and two additional bits representing an attach right and a detach right. Only if the attach right of a particular key k is set, is the process holding this key able to attach a copy of k's value (but not k's rights) to any value v in his (the process') possession. Similarly, a process may detach key k from v only if that process is in possession of the same key k and if the detach right of k is set.

A given key may be attached to a value either as a seal or as a signature. Any value may be sealed or signed (or both) with the same key. For example, assume that a process p1 possesses a key k1 (with both attach and detach rights set) and all other processes hold the same key k1 but with only the detach right set. If any of these other processes pi receives a value v which

has k1's value attached to it as a signature, pi can verify that v originated from process p1 since p1 is the only process able to attach k1. Thus a sender is able to uniquely 'sign' a value as was required under b) above. The requirement b) may be satisfied in a similar fashion. Assume that a process p2 is in possession of a key k2 (whose attach and detach rights are set) and all other processes possess k2 but with only the attach right set. This configuration will gurantee that any process may 'seal' a value v by attaching k2 to it, but the only process able to detach k2 is the process p2. We now devote our attention to explaining how a key attached to a value v as a seal protects v and prevents its contents from being misused. Similarly, we will show how attaching a key as a signature guarantees that the signed value could not have been modified on the way to its intended destination.

As mentioned before, our system is modeled as a collection of processes capable of holding and exchanging copies of pieces of data. (We emphasize again that sharing the same piece of data is explicitly prohibited - a separate copy is given to each process if it is needed.)[*] We further enclose these processes within a 'sphere' representing the <u>boundary of the system</u>. The users of the system are standing outside the sphere and may communicate with its interior only via special windows in the sphere's wall called <u>information disclosure interfaces</u> (IDIs).

---

[*]A separate copy exists, of course, only at the conceptual level; for reasons of efficiency an actual (physical) copy need not be created but any mechanisms implementing possible sharing must be invisible to the user.

has k1's value attached to it as a signature, pi can verify that v originated from process p1 since p1 is the only process able to attach k1. Thus a sender is able to uniquely 'sign' a value as was required under b) above. The requirement b) may be satisfied in a similar fashion. Assume that a process p2 is in possession of a key k2 (whose attach and detach rights are set) and all other processes possess k2 but with only the attach right set. This configuration will gurantee that any process may 'seal' a value v by attaching k2 to it, but the only process able to detach k2 is the process p2. We now devote our attention to explaining how a key attached to a value v as a seal protects v and prevents its contents from being misused. Similarly, we will show how attaching a key as a signature guarantees that the signed value could not have been modified on the way to its intended destination.

As mentioned before, our system is modeled as a collection of processes capable of holding and exchanging copies of pieces of data. (We emphasize again that sharing the same piece of data is explicitly prohibited - a separate copy is given to each process if it is needed.)* We further enclose these processes within a 'sphere' representing the <u>boundary of the system</u>. The users of the system are standing outside the sphere and may communicate with its interior only via special windows in the sphere's wall called <u>information disclosure interfaces</u> (IDIs).

---

*A separate copy exists, of course, only at the conceptual level; for reasons of efficiency an actual (physical) copy need not be created but any mechanisms implementing possible sharing must be invisible to the user.

Any data may leave the system only via an IDI, and then only if it is unsealed. Only after all seals have been removed will the data be allowed to pass an IDI and reach the 'outside world'. Since information inside a computing system is not merely being stored but is also being used for computation to yield new pieces of information, we introduce the following rule: Any value derived from data sealed with a key must inherit the same key as a seal and thus be subject to the same restrictions as the original data. The basic philosophy of our approach then may be summarized as follows:

> <u>A piece of data potentially may propagate to any place within the system. When sealed with a key it is guaranteed that this data and any information derived from it will not be able to leave the system unless the seal is removed.</u>

Since our approach departs significantly from those taken in other systems where data is prevented from propagating unless certain access or capability conditions are met, the following discussion is intended to further explain our point of view.

In no existing system known to the author is a distinction made between the (human) user and the process running under his command. Implicit in such systems is the notion that information accessible to the process is automatically available to the user. Consequently, the secrecy of information must be considered already compromised when it reaches the user's process, and not only after it reached the user himself. We argue that this condition is an unnecessary constraint. Imagine

a sealed box containing secret information. If this box cannot be opened by a 'spy' no disclosure of information will take place even if the spy is actually in possession of the box. Similarly, in a computing facility it is not really the process that must be prevented from illegally accessing sensitive information, but rather the user running that process.

> A process which possesses secret information but which is unable to reveal that information constitutes no danger with respect to protection.

To further illustrate the basic philosophy of our approach we would like to contrast our system with a capability based system such as HYDRA /CoJe75/. There a subject can make use of an object (e.g. read and output a file) if the subject is in possession of a capability for that object. A capability consists of a pointer to the object and a set of rights (e.g. a read right) which determines those operations the holder of the capability may perform on the object.

In order for a subject to make use of an object (e.g. a file) in our system, the subject must necessarily be in possession of that object itself, and in case this object is carrying a set of keys (seals) the subject must also be in possession of all the keys included in the set. Only then is the subject able to unseal and thus utilize the object. We wish to stress that under 'utilize an object' we understand outputing that object (or any information derived from it) to the outside of the sphere. The subject is of course free to perform arbitrary computations with the object since all sealed results will remain within the sphere.

From the above discussion it follows that the purpose of a seal key is to prevent 'leaking' of information. In contrast, the purpose of a signature key is to permit the origin of the signed value to be verified. We do this by guaranteeing that no value produced as a result of any computation involving a signed value will inherit the signature of the original value, save for computations known to induce only the identity transformation. Note that this rule is just the converse of the rule regarding seals which requires that all result must inherit the seals of the original values.

Even though our model is independent of any particular language or machine architecture it was especially developed for applicative languages such as pure LISP, FFP /Bac78/, and in particular, dataflow /ArGoPl78/. We will attempt to justify the usefulness of our approach by giving solutions to several well known protection problems in section 4.

3. Implementation of the Model in a Dataflow System.

3.1. Basic Dataflow Principles.

A primary motivation for studying dataflow is the advent of LSI technology which makes feasible the construction of a general-purpose computer comprising hundreds, perhaps thousands, of asynchronously operating processors /GoTh79/. The semantics of a dataflow program are such that it is implicitly partitioned into small tasks called activities that may be executed asynchronously by independent processors. In this way many processors may cooperate in completing the overall computation.

In our dataflow system, programs are written only in the high-level language Id (for Irvine dataflow). An Id program is compiled into a corresponding program in the base language -- an ordered graph consisting of actors (operators) interconnected by lines that transport values on tokens. For example, Fig. 1 is an expression in Id and Fig. 2 is its compiled form. The execution of every actor is completely data-driven which means that execution is carried out when and only when all operands needed by that activity have arrived. The resulting output values are then sent to other actors which expect those values as inputs. Thus the multiply actor in Fig. 2 will produce the result x*c and send it on a token to the plus actor after having received both the operand x produced by the subtract actor and the operand c (an input to the program).

In addition to the asynchronous, data-driven style of execution, dataflow is conceptually memoryless. All values are carried by tokens exchanged between actors. Thus all calculations are on values rather than on the locations where those values are kept. This implies a purely functional and side-effect free execution. It also implies that no sharing of data is possible since every actor obtains a separate copy of any value it needs as its input. The absence of memory implies that it is not possible to talk about 'access' to data. All information must be supplied to a program explicitly in the form of arguments. These arguments propagate through the graph constituting the program and the final resulting values are returned to the caller of the program. This principal is crucial to protection: A (borrowed) program can never gain access to (e.g. steal or destroy) any information

which is not explicitly passed to it by the caller as an argument. The possibility of a program gaining access to data private to the caller of the program is referred to as the Trojan Horse Problem, and in our system it is immediately solved by the very principles of dataflow. This is discussed in further detail in section 4.1.

3.2. Dataflow Processes.

Dataflow resource managers were introduced into Id /ArGoPl78/ to provide non-deterministic and history-sensitive functions accessible from different parts of a program. An instance of a resource manager is a dataflow program (graph) enclosed between an entry and an exit actor (Fig. 3). The entry actor receives all arguments (e.g. arg1, arg2) sent to that manager, possibly from different users, and forms a stream of tokens directed into the managers body. The body of a manager may be any dataflow expression with a stream argument and stream result. In Fig. 3 we have presumed the body to be a loop which recomputes an 'internal state' on each iteration, where an itteration occurs essentially upon the arrival of each token in the input stream. By making the 'internal state' on each iteration a function of its previous value and the value of the token just obtained from the input stream, the effect of an internal memory is 'artificially' achieved. In this manner the output of a manager may be made to depend upon the history of previous inputs. The stream of result tokens (e.g. res1, res2) is then sent to the exit actor which returns the individual tokens comprising the output stream to the corresponding callers.

In order to be able to call a particular manager, the user must be in possession of a pointer to the manager's domain, m. In Id a call to manager m is denoted

res1 ← use(m,arg1) .

The value m points to the desired manager instance and is supplied together with the argument value arg1 to the primitive use. This primitive sends the value arg1 to the entry of the manager instance and receives the value res1 returned from the manager's exit as the result of m's processing argument arg1. (Similarly for the other use of m in Fig. 3).

We define a process to be an instance of a resource manager. The only way for processes to communicate is by explicitly calling one another through the use primitive. Thus information is always passed explicitly in the form of arguments and results; information is never passed by granting 'access' to the information, (e.g. a portion of memory) as is the case in conventional systems.

Although the above description is in terms of dataflow, it of course holds for any system satisfying the basic properties of no shared data and communication through copied messages. Thus the system described here might be used on a conventional system at the level of processes, where tokens are the messages in an interprocess communication facility.

3.3 Implementation of Protection Mechanisms.

All processes in the system capable of holding and exchanging information are implemented as dataflow resource managers. The

information to be exchanged may be any type of value, e.g. integers, reals, strings, structured values, procedure definitions, etc. The following extensions are introduced in order to implement the protection mechanisms described earlier.

a) A special facility called the <u>key generator</u> is introduced. Only the key generator is able to create (upon request from some process) values of type key. The fact that keys are of a distinct data type is used to eliminate the possibility of (accidentally or intentionally) forging a key: once created a key may never be modified, nor may a new value of type key be produced (other than by the key generator).

b) Initially every key k is obtained from the key generator with both the attach and detach rights set. Each of these rights may be explicitly reset (i.e. set to zero) by the programmer with the two primitives <u>reset-attach-right</u>(k) and <u>reset-detach-right</u>(k) which produces a copy of k with the corresponding right set to zero. For example the statement

k' ← <u>reset-attach-right</u>(k)

produces a copy k' of k with the attach bit set to zero. The detach bit remains unchanged.

c) Every value v carries with it two (possibly empty) sets of keys. We denote this as v{s1,s2,...,sn}{g1,g2,...,gm}.

Each of the keys si is called a <u>seal</u> and each of the keys gj is called a <u>signature</u>.

d) Four primitive operations are defined for attaching keys to and detaching keys from values:

1) v' ← <u>sign</u>(v,k)

The value v' is a copy of v, and in addition the key k is included in the set of signatures of v'. (Only the value but not the rights of k are written. The operation will be successful only if the attach right on k is set, otherwise v' will be an error value.

2) v',f ← <u>unsign</u>(v,k)

In case the key k is an element of the set of signatures carried by v the value v' is a copy of v with the key k removed, and the value f (serving as a flag) is set to true. In case k is not carried by v as a signature then v' is an exact copy of v and f has the value false. Thus the value of f indicates whether the key k was carried by v as a signature. The operation will be successful only if the detach right on k is set, otherwise v' and f are error values.

3) v' ← <u>seal</u>(v,k)

The value v' is a copy of v, and in addition the key k is included in the set of seals of v'. The operation will be successful only if the attach right on k is set, otherwise v' will be an error value.

4) v',f ← <u>unseal</u>(v,k)

Similar to <u>unsign</u>, the <u>unseal</u> primitive gives v' the value of v with the key k removed from the set of seals (in case

this key was present). The value of f then indicates whether the key was present or absent. The operation will be successful only if the detach right on k is set.

e) A primitive operation test-seal(v) is defined on any value v. This primitive may be used by the programmer to detect whether a value is protected by at least one seal. If this is the case the boolean value true is returned, otherwise the value false results.

f) The result of any function f in the system, other than unseal, unsign, and test-seal, must carry the set union of all seals carried by the individual values involved in the computation. An example of a primitive function f is shown in Fig. 4 where the set of seals carried by the result z is the union of sets $\{k1,k2\} \cup \{k1,k3\} = \{k1,k2,k3\}$ carried by the inputs x and y, respectively. As opposed to seals, a signature on an input value v will be inherited by the result of a computation f(v) only if f is an identity primitive. As an example of an identity primitive, we mention the primitive actor use which sends a message (unchanged) from one process to another.

g) As described in section 2, communication between a user standing outside the system sphere and a process running inside is possible only via an information disclosure interface (IDI). An IDI is a process so validated by the systems programmer (i.e., it is trusted). A system properly configured would insure that any piece of data sent to a user terminal must move via an IDI. Each IDI employs the

primitive test-seal defined in e) to test whether the received data is unsealed. If this is the case the IDI forwards the data, thus passing the data through the window to outside the sphere. If a seal is detected the data is not allowed to pass and instead an error message is forwarded. Thus no protected value is able to escape from the system. It is important to relize that the IDIs must be physically interposed between any sending process inside the sphere and a receiving terminal outside the sphere. Thus the IDIs define the boundary of the system - the sphere. In the sequel we will demonstrate the use of the protection system as defined in a) through g) by applying it to two major problem domains: proprietary services and private interprocess communication.

4. Application of the Protection System.

In section 1 we argued that the use of proprietary services may be viewed as a special case of interprocess communication. In the following we concentrate first on the problem of proprietary services and later extend our consideration to include the more general case of interprocess communication.

4.1. Proprietary services.

Most users of a computer system have the need or desire to build on the work of others by utilizing programs and systems provided by other programmers. We will refer to such programs

as (proprietary) services*. Several important protection problems must be solved in order to satisfy the needs of the lessors (owners, providers) and the lessees (users) of such services. The lessee's major concerns are the following:

a) The service must not be able to steal or destroy information which the lessee did not explicitly supply to the service for such purposes. Each such service is employed by sending it the necessary arguments via the <u>use</u> primitive described in section 3.2. Since this is the only way a process can receive information it is guaranteed that a service cannot obtain or destroy information belonging to the lessee or some other process unless that information was explicitly supplied as an argument. Thus the very principles of dataflow solve this problem, referred to as the Trojan Horse Problem.

b) The service must not be able to disclose sensitive information supplied to it by the lessee, but it should be allowed to disclose non-sensitive information, for example for the purposes of billing. The following section (4.2) presents a solution to this problem, usually referred to as the Selective Confinement Problem.

---

*In this paper we assume that all proprietary services are implemented as dataflow resource managers, i.e. separate processes. However, there is no qualitative difference in the results if such services are instead supplied as procedures. The protection primitives previously described are still sufficient.

On the other hand, the lessor's major concerns are the following:

a) The lessee must not be able to destroy or steal (copy) parts of the service. This includes not only the code itself but also any intermediate results that could be misused to deduce information about the principles and methods employed by the service.

b) Permission to use the service must not provide a way for the lessee to steal or destroy information which is not part of the service.

In Id, in order to employ a service only the pointer to the manager (which is the implementation of that service) need be given to the lessee. The only operation defined on such a pointer is the use primitive that communicates the necessary arguments and results between the lessee and the service as was described in section 3.2. These points imply the solution to the above two problems a) and b).

4.2. The Selective Confinement Problem.

The Problem.

The essence of the problem is to guarantee that a borrowed program, e.g. a service routine, will not disclose any sensitive information passed to it by a caller for processing.

Assume, for example, that the lessor provides a proprietary service called Tax which calculates the income tax for any lessee that supplies to it the necessary information, such as salary, deductions, address, etc.. In order to employ the service the

following call must be performed

res ← use(Tax, data)

where 'Tax' is the pointer to the service process, 'data' represents the collection of values supplied by the lessee, and 'res' is the income tax calculated by 'Tax' according to 'data'. Since the lessee of the service may not trust the Tax program, he wishes to prevent certain sensitive information (e.g. the salary) from being disclosed to other users, including the lessor of the service. In addition to computing the income tax the service needs to calculate a bill for the services rendered and to give a copy of the bill to the lessor.

The Solution.

In order to solve the problem the lessee is asked to partition the data sent to the service into two parts - one part contains sensitive information, such as the salary, while the other part contains information not needing protection from disclosure and which is required by the Tax system, e.g. the lessee's name and address, for purposes of billing. In calling the service the lessee may protect the sensitive part of the data by attaching to it a key k known only to the lessee. The non-sensitive part is left unprotected. The call then has the form

sd' ← seal(sd,k);

res ← use(Tax,<sd',fd>);

where sd is the sensitive datapart, sd' is a copy of sd with the key k attached to it, and fd is the non-sensitive (free) data part. (The angle brackets indicate a list of arguments). The flow of information is shown in Fig. 5. The service computes the result and returns it to the lessee as the value res. For

example, a computation within the service process might be

r ← compute_tax(sd',fd)

where the value r will inherit the key k from sd'. If r is the value returned (possibly at some later point) to the lessee as res (by the use primitive shown above), he is able to detach the key k from res by

res' ← unseal(res,k)

and output the unsealed value res' - the income tax. On the other hand the bill may be computed by the service using only the non-sensitive data, as in

bill ← compute_bill(fd) .

If this is the case the value bill will be unsealed, and if sent to the lessor for subsequent billing of the lessee it may be disposed of freely, that is it may be output. The value r (res) and any other values possibly derived from sd' are sealed with the key k. Hence, even if sent to the lessor by the service, these values cannot be utilized since no information disclosure interface will permit these values to leave the system. We presume, of course, that the lessee did not give k (directly or indirectly) to the lessor. Note that we do not prevent the service from propagating any of the sensitive data to other processes. This permits the service to employ yet other services on its own behalf.

4.3. Private Interprocess Communication.

Our protection system is centered around the following two facilities:

a) A process m1 sending a sealed value v to a process m2 is guaranteed that only m2 will be able to unseal and

utilize v, even if v was forwarded through a number of 'courier' processes. Assuming that some of these processes cannot be trusted, it is not possible to guarantee that v will reach m2. However, it is always guaranteed that no information contained in v can ever be misused by other processes.

b) Process m1 signs v before releasing it to any 'courier' process. Thus m2 may verify the authenticity of v and prevent any other process from masquerading as m1.

The mechanisms behind the above facilities are based on the idea of trapdoor functions presented in /RiShAd78/. A trapdoor function f has the property of possessing an inverse function f^c not easily derivable from f. The function f is made public and may be used to encode messages that can be decoded only by the user who is in possession of f^c. On the other hand the function f^c provides an unforgable 'signature' on messages originating from the user in possession of f^c (since this is the only user in possession of f^c). In our system we do not encode messages in a cypher, rather we use keys as signatures and seals that are attached to the message. Initially every process that wishes to engage in communication with another process obtains a new key from the key generator. Assume that process m1 is to send messages to m2. For this purpose m1 will make a key k1 (obtained from the key generator) public, but with only the detach right set. By "making public" we mean to advertise or otherwise disclose (for example, to a user information service) so that any process, in particular m2, can obtain it. Similarly, process m2 will make its own key k2 public,

but with only the attach right set. From the above it follows that the key k1 can be attached only by m1 but may be detached by any process in the system. Conversely the key k2 may be attached by any process, but can be detached only by m2. If m1 is to send a message m to m2, m1 attaches the key k1 as a signature and the key k2 as a seal to m:

m' ← <u>sign</u>(m,k1);

m" ← <u>seal</u>(m',k2)

Both operations will be successful since m1 has k1 with both rights set and k2 with the attach right set. The message m" will then have the form m"{k2}{k1}. When m" is received by m2, m2 may remove both keys as follows:

m',f1 ← <u>unsign</u>(m",k1);

m,f2 ← <u>unseal</u>(m',k2)

The above attach/detach operations have the following consequences:

a) The received message m" is sealed with the key k2. Only m2 is in possession of k2 with the detach right set, hence only m2 is able to successfully perform the <u>unseal</u> operation which yields the unsealed value m. This gives m1 the guarantee that only m2 will be able to utilize the information contained in m".

b) The flag f1 may be used by m2 to detect whether the key k1 was carried by the message as a signature. If this is the case (f1 is true) then m2 is certain the message originated from m1, since m1 is the only process in possession of k1 with the attach right set and thus the only

process that could have signed the message.*

From the above it follows that by using the same mechanisms as those employed to solve other problems (sections 4.1, 4.2) it is possible to establish a communication between processes such that both communication partners are certain to exchange messages with the desired process and no 'spying' or 'masquerading' can take place. Of course, the system is not limited to just pairs of communicating processes; two groups of logically equivalent senders and logically equivalent receivers may communicate with equal assurance of protection.

5. Conclusions.

This paper presented a protection mechanism which is simple to understand and to use, yet powerful enough to allow the solution of a large variety of protection problems. The entire mechanism presented here is based on attaching and detaching unforgable keys to values and is controlled by the programmer through only a few primitives. Despite its simplicity we have been able to give solutions to problems which cannot be solved easily in most other advanced protection systems, e.g. the Selective Confinement Problem.** The price paid for the capabilities of the

---

*The flag f2 is not used in this example. It may be employed to detect whether a particular key was carried by a value as a seal.

**Further application examples may be found in /Bic78/ where we present solutions to problems such as the "Prison Mail System" /AmHo77/, "Sneaky Signaling" /Lam73/, /Rot74/, and problems related to file systems.

system is overhead in computation: for every primitive operation some computation producing a new set of keys might be necessary. However, since the computation of the new set of keys is independent of the computation of the actual value, a processing unit can be designed to perform both tasks in parallel. Thus little degradation of the actual computational performance need be introduced due to the protection mechanism. With decreasing cost of hardware the cost of the additional processors or processor components would appear minimal. This argument holds especially in the case of a dataflow machine which consists of a large number of inexpensive processors available through LSI technology.

Acknowledgements.

I would like to express my sincere thanks to Prof. K.P.Gostelow (UC Irvine) and Prof. Arvind (M.I.T.) for reviewing and commenting upon the countless drafts of this paper.

```
(x ← a - b;
 y ← x * c
 return y + x)
```

Figure 1

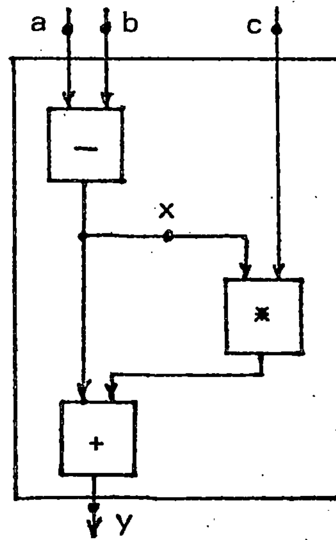

Figure 2

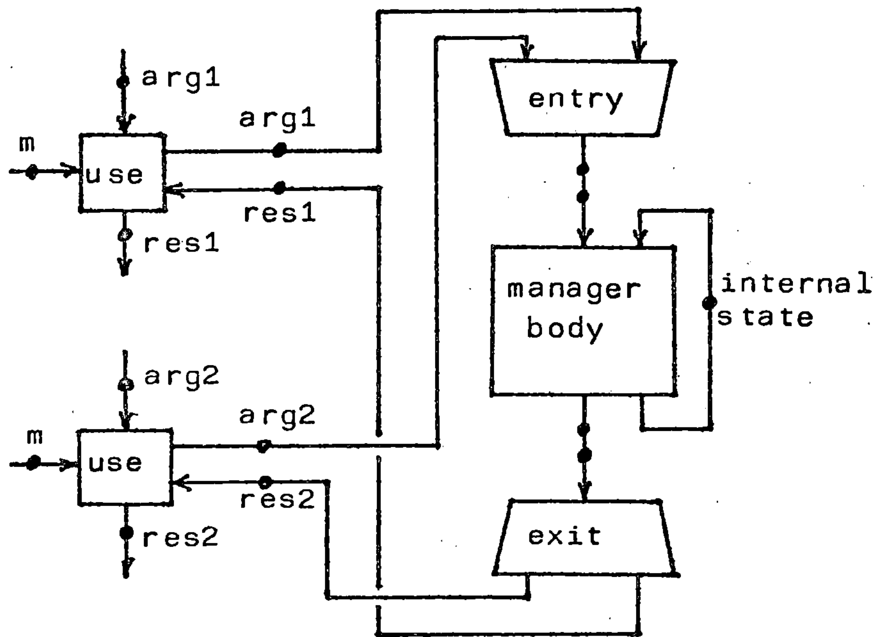

Figure 3

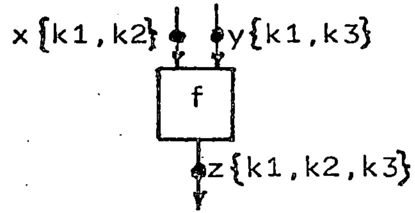

Figure 4

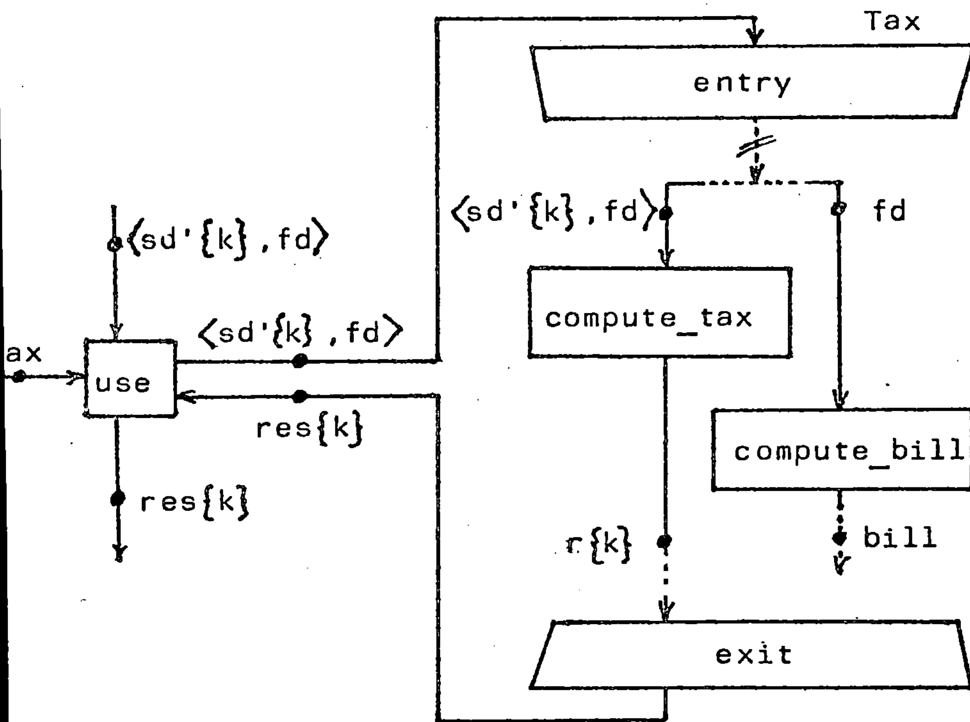

Figure 5

REFERENCES:

/ArGoP178/ Arvind, Gostelow,K.P., Plouffe,W. An Asynchronous Programming Language and Computing Machine. TR-114 Dept. of ICS, UC Irvine, Ca 92717

/AmHo77/ Ambler,A.L., Hoch,C.C. A Study of Protection in Programming Languages. SIGPLAN Notices, Vol.12, Nr.3, March 77

/Bac78/ Backus,J. Can Programming be Liberated from the von Neuman Style?. CACM, August 78.

/Bic78/ Bic,L. Protection and Security in a Dataflow System. Ph.D. Thesis, Dept. of ICS, UC Irvine, CA 92717. 1978

/CoJe75/ Cohen,E., Jefferson,D. Protection in the HYDRA Operating System. SIGOPS, Nov. 1975.

/GoTh79/ Gostelow,K.P., Thomas,R. Performance of a Dataflow Computer. TR-127A, Dept.of ICS, UC Irvine, CA 92717. Submitted for publication. 1979

/Jon73/ Jones,A.K. Protection in Programmed Systems. Ph.D. Thesis, Carnegie-Mellon University. 1973.

/Lam73/ Lampson,B.W. A Note on the Confinement Problem. CACM, Vol.16, Nr.1o. October 1973.

/RiShAd78/ Rivest,R.L., Shamir,A., Adelman,L. AMethod for obtaining Digital Signatures and Public-Key Cryptosystems. CACM, Vol.21, Nr.2. February 1978.

/Rot74/ Rothenberg,L.J. Making Computers Keep Secrets. Ph.D. Thesis, MAC-TR-115, M.I.T., Cambridge, Mass. 1974.

/Sch72/ Schroeder,M.D. Cooperation of Mutually Suspicious Subsystems. Ph.D. Thesis, MAC-TR-1o4, M.I.T., Cambridge, Mass. 1972.