# CONFINEMENT OF INFORMATION IN A DATAFLOW SYSTEM

by

Lubomir Bic

Technical Report #177

Department of Information and Computer Science
University of California
Irvine, California 92717

March 15, 1981

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

## ABSTRACT

A protection model is presented for a multi-user dataflow computing system which is incorporated into its functional high-level language. The model is based on tags attached as 'seals' to values exchanged among processes to prevent leaking of information. A tag 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 cannot leave the system unless a matching tag is presented. Any function applied to sealed values will produce results that carry the union of all seals carried by the argument values. Thus, it is also guaranteed that no information derived from a sealed value will be able to leave the system unless it is explicitly unsealed.

The functioning of the system is demonstrated by giving solutions to well known protection problems, for example from the area of proprietary services, such as the 'Selective Confinement Problem' and the 'Trojan Horse Problem.'

## 1. Introduction and Objectives

In recent years the need for better and less restrictive protection mechanisms has emerged. A variety of advanced protection systems have been proposed but the drawback most common in such systems is their complexity in both use and understanding. In addition, many well known protection problems still have no satisfactory solution in these 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 functional high-level language. Thus the implementation and enforcement of protection policies do not require that the user leave the domain of the language in which his programs are written.

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 area of proprietary services, such as the 'Selective Confinement Problem' /Lam73, DeGr74/ and the 'Trojan Horse Problem' /Sch72/.

## 2. An Intuitive Description of the Model

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. 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.*

All processes are enclosed within a single 'sphere' representing the boundary of the system. 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 information disclosure interfaces (IDIs). Data may enter and leave the system only via an IDI.

## 3. A Functional View of Protection

Our model is concerned with controlling information dissemination. We provide machanisms to allow any value to be protected by attaching to it a unique tag referred to as a seal. A seal will prevent that value (and any information derived from it) from leaving the system (the sphere). Seals may be used in only well defined ways which prevent

---------------

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

user processes from forging or otherwise compromising the seals of other processes as will be discussed in Section 4.3.

Conceptually, any number of seals may be attached to a value v (possibly by different processes). A function f performed on the value v produces a new value v' as a result. We require that v' inherit _all_ seals carried by the value v. In case more than one input value is required by the function, the resulting value v' must inherit the union of all seals carried by the input values. This may be expressed as follows:

```
v'<- f(vl, v2, ..., vn)
seals(v') <- seals(vl) U seals(v2) U ...  U seals(vn)
```

where vl,v2, ..., vn are the input values to f and seals(vi) is the set of seals carried by the value vi.

_Thus at any stage of the computation the protection of a value v may be expressed as the set of seals seal(v) computed as the union of all seal sets carried by the input values._

The purpose of attaching seals to a value is to prevent leaking of information contained in that value. As stated above, any data may leave the system only via one of the windows (IDIs) in the sphere. At the point of output the protection of a value is examined. Only values which have an empty set of seals will be allowed to pass the IDI and reach the "outside world". Since no function applied to a sealed value v can produce a result which is less protected than v itself, it is guaranteed that no leaking of information derived from v can take place, regardless of the functions used.

The basic philosophy of our approach may then be summarized as follows:

> A piece of data potentially may propagate to any place within the system. When protected with a seal 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.

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 most existing systems known to the author no distinction is made between the (human) user and the process running under his command. Implicit in such systems is the notion that the information accessible to a user's process is automatically available to that user. Consequently, the secrecy of information must be considered compromised as soon as it reaches an unauthorized user's process. We argue that this condition is unnecessarily restrictive. 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.

> <u>A process which posseses secret information but which is unable to reveal that information constitutes no danger with respect to protection</u>.

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/. In HYDRA a process can make use of an object (e.g. read and output a file) if the process 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 process to make use of an object (e.g. a file) in our system, the process must necessarily be in possession of that object itself, and in case this object is carrying a set of seals the process must also be in possession of all the seals included in the set. Only then is the process able to unseal and thus disclose the object outside the sphere.

## 4. Implementation of the Model in a Dataflow System

Even though our model is independent of any particular language or machine architecture it was especially developed for functional systems such as dataflow /ArGoPl78, Den73/. We will attempt to justify our approach by giving solutions to several well known protection problems in Section 5.

### 4.1 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 procesors may cooperate in completing the overall computation. In the dataflow system developed at the University of California, Irvine, 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 -- a directed 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 actor 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 contents of addressable memory cells. This implies that no sharing of data is possible since every actor gets a separate copy of each input value. The absence of memory implies also that it is not meaningful to talk about 'accessing' data. All information must be supplied to actors and collections of actors (e.g. expressions, procedures, programs) 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 principle is crucial to the protection system described here: No program can ever 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 fundamental principles of dataflow. This is discussed in further detail in Section 5.1.

## 4.2 Dataflow Processes

Dataflow managers were introduced into Id /ArGoPl78/ to provide the non-determinism necessary for managing resources such as airline reservation databases, etc.. An instance of a 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 a stream result. In Fig. 3 we have presumed the body to be a loop which recomputes an 'internal state' on each iteration which 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 <u>internal</u> <u>memory</u> is 'artificially' achieved. In this manner the output of a manager may be made to depend on 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 reference m to the instance of the manager. In Id a call to the manager m is denoted as:

```
     res1 <- use(m,arg1)
```

The value m refers to the desired manager instance and is supplied together with the argument value arg1 to the primitive actor <u>use</u>. 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 its processing argument arg1. (Similarly for the other use of m in Fig. 3).

We define a <u>process</u> to be an instance of a manager. The only way for processes to communicate is by explicitly calling one another through the <u>use</u> primitive. Thus information is always passed explicitly in the form of arguments and results; information is never passed by granting 'access' to 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 which communicates only 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 inter-process communication facility.

## 4.3 Implementation of Protection Mechanisms

All processes in the system capable of holding and exchanging information are implemented as dataflow managers. The information to be exchanged may be of any type, 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 seal generator is the only facility capable of creating values of type seal. The fact that seals are of a distinct data type is used to eliminate the possibility of (accidentally or intentionally) forging a seal: once created a seal may never be modified, nor may a new value of type seal be produced other than by the seal generator.

b) Every value v carries with it a (possibly empty) set of seals. We denote this as v{s1,s2,...,sn}.

c) Two primitive operations are defined for attaching seals to and detaching seals from values:

1) v' <- _seal_(v,s)
The value v' is a copy of v, and in addition the seal s is included in the set of seals of v'.

2) v',f <- _unseal_(v,s)
In case the seal s is an element of the set of seals carried by v then the value v' is a copy of v with the seal s removed and the value f (serving as a flag) is set to true. In case s is not carried by v as a seal then v' is an exact copy of v and f has the value false. Thus the value of f indicates whether the seal s was carried by v.

d) 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_ is returned.

e) The result of any function f in the system, other than _unseal_, 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 {s1,s2,s3} carried by the result z is the union of the sets {s1,s2} and {s1,s3} carried by the inputs x and y, respectively.

f) As described in Section 2, communication between a user standing outside the system sphere and the processes inside is possible only via an information disclosure interface (IDI). Each IDI is a special process which is validated by the administrator of the system (i.e. it is trusted). A system properly configured would ensure that any piece of data sent to a user terminal must move via an IDI. Each IDI employs the primitive _test-seal_ defined above to test the received data. If that data is unsealed the IDI passes it through the window to the outside. If a seal is detected the data is destroyed and an error message is returned to the process attempting to use the IDI. Thus no sealed value is able to escape from the system. It is important to realize 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 f) by applying it to problems associated with providing proprietary services.

## 5. Application of the Protection System

### 5.1 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. We argue that using 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 (e.g. the arguments) to the service which 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. 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. Each such service is employed by sending it the necessary arguments via the <u>use</u> primitive described in section 4.2. 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 fundamental principles of dataflow solve this problem, referred to as the Trojan Horse Problem.

b) The service must not be able to disclose sensitive information suplied 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 (5.2) presents a solution to this problem, usually referred to as the Selective Confinement Problem.

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 informaton which is not part of the service.

In Id, in order to employ a service only a reference to the manager (which is the implementation of that service) need be given to the lessee. The only operation defined on such a reference is the use primitive that communicates the necessary arguments and results between the lessee and the service as was described in Section 4.2. These points imply the solution to the above two problems a) and b).

## 5.2 The Selective Confinement 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 reference to the service process, 'data' represents the collection of values supplied by the lessee, and 'res' is the income tax calculated by 'Tax' based on '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. On the other hand, the service, in additon to computing the income tax, needs to calculate a bill for the services rendered and to give a copy of the bill to the lessor.

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, e.g. the lessee's name and address, which is required by the Tax system for the purpose of billing. In calling the service the lessee may protect the sensitive part of the data by attaching to it a seal known only to the lessee. The non-sensitive part is left unprotected. The call then has the form

```
     sd' <- seal(sd,s);
     res <- use(Tax,<sd',fd>);
```

where sd is the sensitive datapart, sd' is a copy of sd with the seal s 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 seal s from sd'. If r is the value returned (possibly at some later point) to the lessee as res (by the <u>use</u> primitive shown above), he is able to detach the seal s from res by

```
     res' <- unseal(res,s)
```

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 used freely, that is, it may be output. The value r (res) and any other values possibly derived from sd' are sealed with the seal s. 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. 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.

## 6. <u>Conclusions</u>

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 seals 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 system is overhead in computation: for every primitive operation some computation producing a new set of seals might be necessary. However, since the computation of the new set of seals 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.

*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.

## Acknowledgements

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

REFERENCES:

/ArGoPl78/ Arvind, Gostelow, K.P., Plouffe, W., An Asynchronous Programming Language and Computing Machine; TR-114a, 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

/Bic78/ Bic, L., Protection and Security in a Dataflow System; Ph.D. Thesis, Dept. of ICS, UC Irvine, CA 92717, 1978. Available from University Microfilms 300 North Zeeb Rd, Ann Arbor, Michigan 48106

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

/DeGr74/ Denning, D., Denning, P., Graham, The Selective Confinement Problem; Proc. Int'l Workshop on Protection in Operating Systems, IRIA, 1974

/Den73/ Dennis, J.B., First Version of a Dataflow Procedure Language; MAC Tech. Memorandum 61, May 1975

/GoTh79/ Gostelow, K.P., Thomas, R., Performance of a Simulated Dataflow Computer; IEEE Transactions on Computers C-29, 10, Oct. 1980

/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.10, October 1973

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

/Sch72/ Schroeder, M.D., Cooperation of Mutually Suspicious Subsystems; Ph.D. Thesis, MAC-TR-104, M.I.T., Cambridge, MA, 1972

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

Fig 1: An Id expression

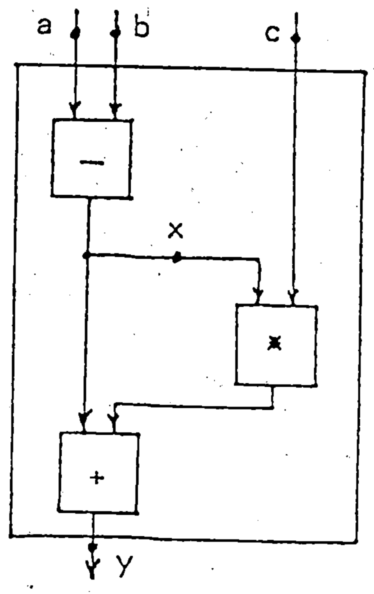

Fig 2: The base language graph corresponding to the expression of Fig 1

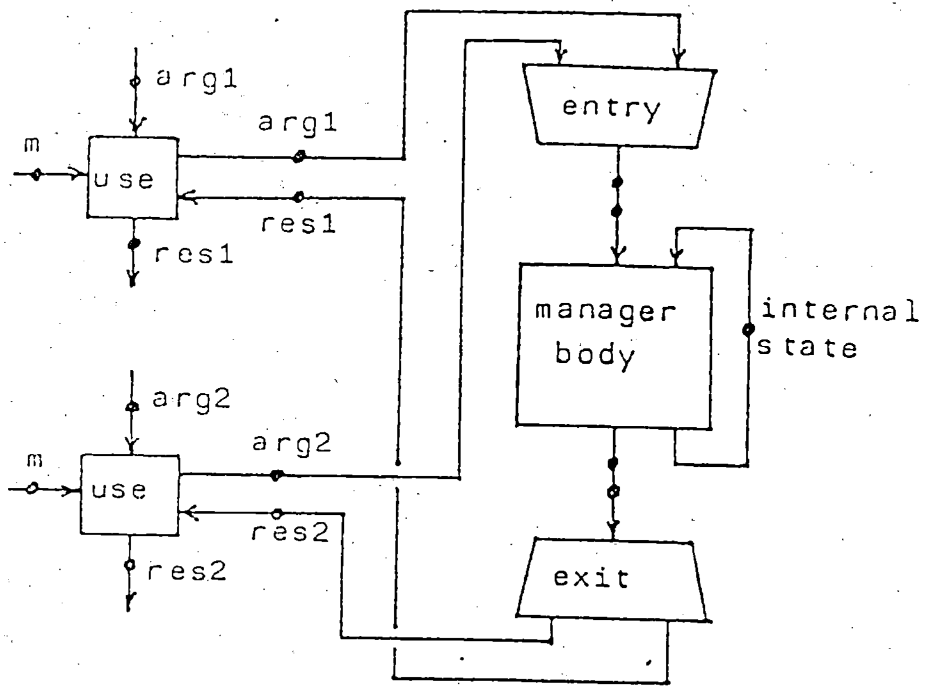

Fig 3: A resource manager and two use actors

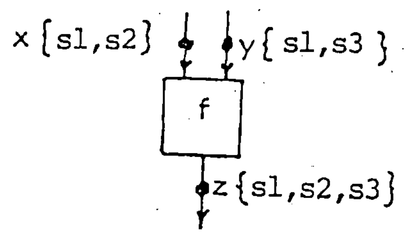

Fig 4: The set of seals on the result of an operation is the union of the input seal sets

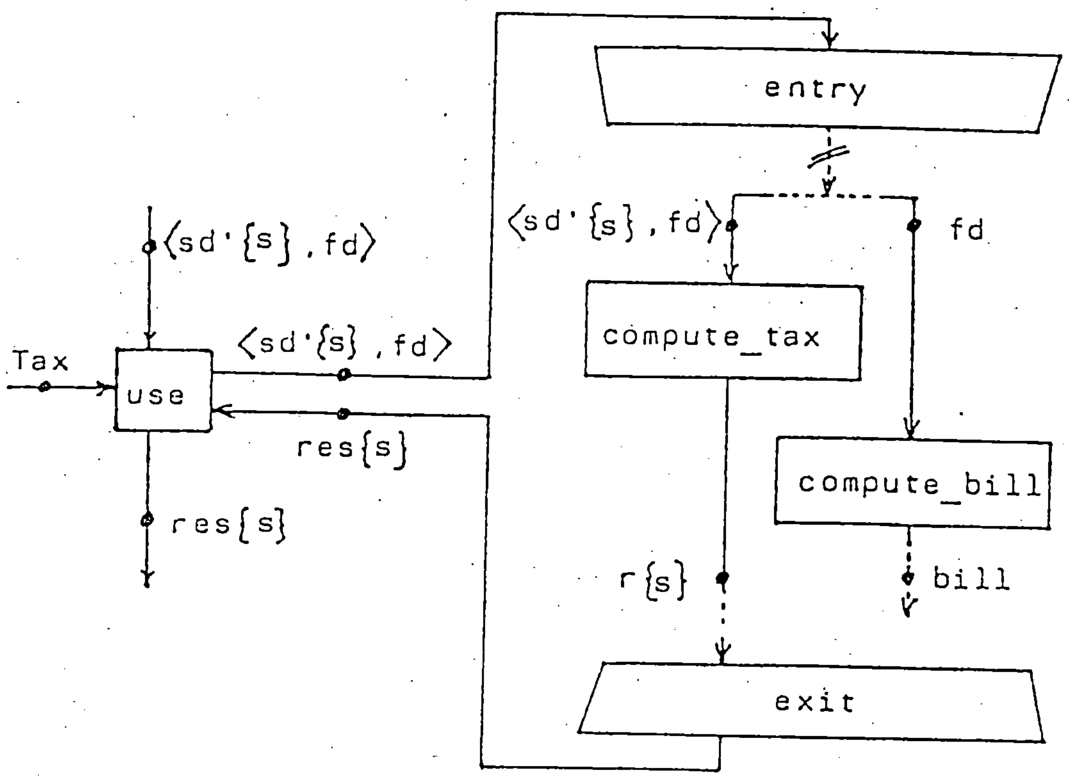

Fig 5: The data sd' is selectively protected from disclosure while it is being used by the Tax program