1974

Selectively Defined Subsystems

Dorothy E. Denning

Peter J. Denning

G. Scott Graham

Report Number:

74-124

SELECTIVELY CONFINED SUBSYSTEMS

Dorothy E. Denning
Peter J. Denning
G. Scott Graham*

Computer Sciences Department
Purdue University
West Lafayette, Indiana 47907

CSD-TR 124

*Department of Computer Science, University of Toronto, Toronto, Ontario M5S 1A7, Canada

NE RIEN INSCRIRE :

Première ligne de texte

NE RIEN INSCRIRE :

SELECTIVELY CONFINED SUBSYSTEMS

Dorothy E. Denning
Peter J. Denning
G. Scott Graham

Purdue University

Abstract: The implementation of programming systems that cannot leak confidential information is examined. Unless severe restrictions are placed on their form, programs of such systems cannot even be permitted to output apparently nonconfidential information unless they have been proved error-free.

Début du texte pour la 1ère page seulement

Introduction

Satisfactory solutions are now known for a variety of protection problems ranging from controlled access to programs and data to mechanisms for debugging subsystems. However, a problem still requiring investigation is the confinement problem; Lampson defines it as the problem of constraining a "service process" so that it cannot leak any information about its "customer processes" [1]. He outlines a solution to the problem, which in essence constrains the service process from retaining any information after it ceases to operate on behalf of a customer process, but it may share information with another process as long as the other process is similarly confined, or else trusted by both the customer and the server. We shall refer to his as the approach of total confinement.

Our purpose here is investigating an approach to the confinement problem based on selective rather than total confinement. A process or subsystem of processes is regarded as being selectively confined if it is free to retain or share information which is not confidential with respect to a customer process, but not information which is; moreover, a customer may declassify previously confidential information for retention by the service. For example, a selectively confined income tax computing service may be allowed to retain address and billing information on its use by customers, but not information on its customers' incomes. This type of problem has been referred to as the cooperation between mutually suspicious subsystems, one of which is "memoryless" [2].

We begin by proposing a mechanism which "obviously" provides selective confinement; however, closer inspection reveals an important limitation in the mechanism. We see no easy way to resolve the limitation, and we are led to the conclusion that, in the current state of the art, no solution to the confinement problem, short of total confinement, is viable.

This work was supported in part by NSF Grant GJ-43176. Authors' present addresses: P.J. Denning and D.E. Denning, Computer Sciences Department, Purdue University, W. Lafayette, Indiana 47907, U.S.A. G.S. Graham, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 1A7 Canada.

NE RIEN INSCRIRE :

Première ligne de texte

General Properties of a Confinement Mechanism

Consider a computing system with processes $P_0,\ldots,P_n$ and data segments $N_1,\ldots,N_m$. Interprocess communication is handled by message sending primitives, such as send message, get message, send reply, and get reply. The segments may be regarded as logical or physical data structures corresponding to files, memory units, registers, etc. and are partitioned into two classes: local and global segments. A segment is local (or private) if it is accessible to exactly one process; otherwise it is global (or shared). Note that two processes with access to the same global segment N_i may be able to communicate by transmitting data via N_i.

Let P_c denote a customer process and P_s a service process which is to operate for P_c in selectively confined mode. Let C denote data considered confidential by P_c; as will be discussed below, the size of C can grow because any data P_s (or a process called by P_s) derives from C will be added to C, and it can shrink in case P_c releases it from confidential status (declassifies it). Listed below are six general properties for a mechanism of selective confinement; though they may seem restrictive, they constitute a minimal set of constraints under which P_c and P_s are likely to agree to operate, given their mutual suspicions.

A central concept below is called engagement. In general, a process P_j is said to be engaged by its caller P_i, whenever P_i sends confidential data to P_j. However, P_i will not be permitted by the system to engage P_j, unless P_j has previously agreed to operate under the rules of selective confinement, and has met all requirements necessary for this mode of operation. We postulate a Boolean system function *certified*(j) which returns true if and only if P_j is certified to have met the requirements for selectively confined operation. Then P_i may engage P_j if and only if *certified*(j), and only if P_j is not already engaged.

In the following, assume that $P_0, P_1, P_2, \ldots$ denotes a system of processes such that $P_0 = P_c$ is the customer, P_1 is the service P_s, P_i for $i > 1$ are processes which can be employed by P_1, and *certified*(i) for $i > 0$. In the sequence, $i < j$ implies that P_i was called earlier than P_j. A single set of confidential data C, initially provided by P_0, is assumed throughout.

1. Mutual Exclusion (one customer at a time). P_j is engaged by P_i as soon as P_i sends P_j a message containing data from the confidential set C, providing that P_j is not already engaged. While P_j is engaged, it may receive confidential data only from its caller, or any processes it engages.

2. Closure. If P_i performs an operation using any data from C, the result of that operation is added to C: Any information derived from confidential data is itself confidential. (Precisely stated, if any of $x_1,\ldots,x_n$ are in C, then the result $f(x_1\ldots,x_n)$ of operation f is added to C.)

3. Non-Leakage. P_i may place an element of C in a segment N only if N is local to P_i (local segments are inaccessible to other processes).

4. Transitivity. If P_i sends a message to P_j ($i\neq 0$) containing data from C, then P_j becomes engaged by P_i. Moreover, P_i may not disengage itself from its caller until P_j disengages itself from P_i. In other words, all processes which eventually receive data from P_0's set C become engaged (effectively by P_0) and must be confined.

5. Declassification. Data may be declassified (removed from C) only by P_0, on receipt of a message from P_1 requesting declassification of data contained in the message. In general, if P_j ($j>1$) wants data declassified, it must request so from its caller P_i ($i<j$); this is repeated by a chain of messages until the original customer P_0 is consulted.

4. Transitivity. The engagement operation must verify that if P_i attempts to engage P_j, then D_j = (0,undefined) and *certified*(j). If this is true, then D_j := (1,i), and j is added to the engagement list L_i of its engagor P_i.

5. Declassification. Postulate a system operation *release*(x) for setting the confidentiality tag of x to 0 without changing the value of x. This operation could be performed only by the process (in this case P_0) which set the tag in the first place; in terms of our model, *release*(x) cannot be executed by any engaged process. If P_j is engaged, it can obtain the release of x only by sending a message to its engagor P_i (i<j), if i≠0, P_i would forward the message to its engagor, and so on until P_0 was contacted. The declassified x would be transmitted back to P_j by a reverse chain of messages.

6. Disengagement. P_j would request disengagement by a system function *disengage*. This function would be allowable only if the engagement list L_j is null, whereupon it would have the effects of a) removing j from the engagement list L_i, where D_j = (1,i), then b) setting D_j to (0,undefined), and c) purging from P_j all elements of C - i.e., any data whose confidentiality tag is set.

Leakage of Confidential Data

Unfortunately, the mechanism we have specified does not prevent leakage of confidential data! Although a confined process P_j cannot directly leak data that is flagged confidential, there is nothing in our mechanism to prevent it from leaking non-confidential data that is equal in value to confidential data. For example, if X∈C and N is a global segment, then the value of X can be leaked by executing the statement

if X = Y *then write* Y *into* N.

Lampson discusses other subtle forms of leakage, such as leakage on "covert channels" (e.g., by cleverly altering the system load) in [1].

In our effort to find a solution to this problem, we made the following observation: Many very subtle examples of leakage can be constructed by embedding statements communicating non-confidential variables in program segments conditioned on Boolean tests on confidential data. A solution to the problem is then briefly stated as follows: Let b be a Boolean expression and A an action conditioned on b. By the closure rule, if b contains an operand X∈C, then b∈C. The problem is then solved by inhibiting all communication by an engaged P_j while P_j is executing A if b is confidential. Hence P_j would not be allowed to write into a global segment or issue spurious messages to another process while it was acting on confidential data.

Isolating the action A, however, involves a complex flow analysis of the code because of the possibility of side effects. Consider, for example, the following statements, where X is confidential and N is a global segment:

if X = 0 *then* Y:=0;
if Y = 0 *then write* Z *into* N.

Here the action "*write* Z *into* N" is indirectly conditioned on the confidential Boolean "X=0". Detecting this involves a flow analysis that takes into account data flow as well as control flow. Such a flow analysis would probably have to be performed on the source code (for efficiency as well as practicality considerations) and the compiler would have to delimit the body of the actions in the machine code. Upon evaluating a confidential Boolean, the hardware (with the possible help of software routines) is then responsible for insuring that all communications [illegible] stopped while executing instructions within the body of the associated action.

A more attractive solution to the problem involves the use of type checking and compile-time certification. Here the programmer declares all variables to be either confidential or non-confidential. The compiler uses this information to

6. Disengagement (and Non-Retention). When P_j disengages from its caller P_i ($i<j$), it is not permitted to retain any data in C; to enforce this, the system will purge from P_j all remaining elements of C as part of the disengagement operation. (If P_j refuses to agree to this, the Mutual Exclusion rule will guarantee the total isolation of P_j from the rest of the system.)

The above rules in fact specify the operation of a selectively confined system of processes, with entry process P_1. The system is the set of all selectively confined processes formed by taking the closure of the transitivity relation suggested by rule 5 (i.e., it is the set of all selectively confined processes that may become engaged data either directly or indirectly by P_0). The elements of the confidential data C are distributed among the processes of the system $P_0, P_1, P_2, \ldots$. The mutual exclusion rule ensures that any confidential data in an engaged process P_i ($i>0$) is a member of the one set C. The closure rule ensures that any data derived in any P_i is added to C. The nonleakage rule keeps elements of C local to each P_i. The transitivity rule provides that each P_i is confined, or communicates only with other confined processes. The declassification rule permits any process P_i to get data removed from C, but only with the explicit permission of P_0. Finally, the disengagement rule guarantees that no element of C remains accessible to P_i when it disengages itself from its caller.

Implementation

Let $P_0, P_1, P_2, \ldots$ denote a system of selectively confined processes with customer P_0 and server P_1. Associate with each process P_i is an engagement list, L_i, containing indices of all processes directly engaged by P_i; initially L_i is null. Associate with each process P_j an engagement descriptor $D_j = (e,i)$, in which at a particular time

- $e = 1$ implies P_j is engaged by P_i, and
- $e = 0$ implies P_j is not engaged and i is undefined.

Associate with each data element a special bit, called the confidentiality tag, set to 1 if and only if that element is in C; this tag can be set to 1 for a datum x by an unengaged process, using a system operation *settag*(x). Then any datum referenced by P_j is considered confidential if and only if it is so flagged. This could be implemented trivially in a tagged architecture [3].

The implementation of the six properties of selective confinement proceeds as follows.

1. *Mutual Exclusion*. Engagement of P_j by P_i is allowable only if D_j = (0,undefined) and *certified*(j). When allowable, engagement has the effect of setting D_j to (1,i) and adding j to the engagement list L_i. The processes P_i and P_j may exchange messages while P_j is engaged by P_i, but P_j may communicate with no other process except those it engages. Engagement is effected by a primitive operation *engage*($P_j; x_1, \ldots, x_n$), where $x_1, \ldots, x_n$ are parameters. Transmission of messages containing confidential data from engaged to unengaged processes is prohibited.

2. Closure. To implement the closure rule we simply tag the result of any operation f that is applied to operands $x_1, \ldots, x_n$ whenever at least one of the x_i is tagged. This is easily handled by hardware in a system with tagged architecture, by ORing the confidentiality tags of the operands to obtain the flag of the result.

3. Non-Leakage. To implement the non-leakage rule we simply raise an error condition if P_j attempts to transfer a tagged datum to a global segment. This can be handled by a supervisor I/O routine (if the global segment is a file, say) or by hardware, in the case of tagged architecture and a segmented virtual memory. The effect of raising the error condition may result in the automatic purging of all confidential data from P_j's memory.

determine which expressions have confidential results. By simple control flow analysis of a program, the compiler examines all statements in the body of each action conditioned on a confidential Boolean: it disallows in them any output statements and gives type errors if nonconfidential variables are assigned confidential results. For example, consider again the program segment

```
if X = 0 then Y:=0;
if Y = 0 then write Z into N;
```

with X declared to be confidential, and Y declared to be non-confidential. Since the expression "X=0" is then known to be confidential, the compiler would detect a type error with respect to Y, and the program would not be certified.

Début du texte pour la 1ère page seulement

This solution is more attractive for two reasons: the flow analysis is simple, and it allows most of the problem to be solved at compile-time. The only check that must be performed dynamically verifies that the actual parameters (or inputs to the program) do not exceed the declared confidentiality of the formal parameters.

Closer scrutiny, however, reveals that the problem is <u>still</u> not solved! For example, consider the following sequence of statements, where X is declared confidential, I is declared non-confidential, and N is a global segment:

```
I:=0; SUM:=0;
repeat
  SUM:=SUM + X;
  I:=I + 1;
  write I into N
forever
```

Since the iteration does not appear to be conditioned on X, the compiler would certify this program segment. Now, suppose the program executes, but after I_0 iterations SUM overflows - i.e., the value of SUM exceeds MAX, the largest number storable in a register. Since the value of I_0 has been put in a global segment, another process can subsequently retrieve it and estimate X from MAX/I_0.

The reason for this problem is that the Boolean expression "SUM overflows" implicitly controls the loop, although it is not explicitly stated. If the programmer had instead written

```
I:=0;SUM:=0
repeat
  SUM:=SUM + X;
  I:=I + 1;
  write I into N
until SUM overflows
```

then the compiler would have detected the type error with respect to I and not certified the program.

The preceding problem arises with all dynamic error conditions, including even software checks on array bounds. This is because all such error conditions represent Booleans that cannot be analyzed at compile-time. We are thus led to our final conclusion: the program must contain no errors! The compiler can safely certify a program for confinement if and only if it can prove the program to be correct. This implies that the compiler must perform range checking as well as type checking. Hence, the programmer must specify a range of values for each input parameter. At execution time, the system must also verify that the values of the actual parameters fall within the range of the formal parameters.

Another possible approach is to permit a program to execute without certification beyond the type checking mentioned earlier. Then if an error should result during execution of the program, the owner of the confidential data would have the opportunity to sue for breach of confidentiality. In order to prove whether or not the program had leaked data, a trace of the confined program's outputting behavior is required, which trace would automatically be transmitted to the customer if the service generated an error. The court must then be able to examine this trace as well as the program code. In the long run, it would be cheaper for services to provide programs whose correctness can be verified.

The foregoing discussion has shown that enforcement of the proposed Non-Leakage Rule (an engaged process may output only nonconfidential data) is considerably more difficult than superficial consideration might lead one to believe. In the present state of the art, the only feasible Non-Leakage Rule is: An engaged process may not under any circumstances write into a global segment or communicate with a nonengaged process, and all data it has written into local segments - except for declassified data - must be purged on disengagement if an error has occurred anywhere in the confined system. Under this rule the mechanism we have proposed is an implementation of Lampson's totally confined system, with the following exceptions: Data declassified by the customer may be retained in the local segments of a process after disengagement, and other non-confidential data may be retained if no errors have occurred.

Conclusions

The mechanism of selective confinement described in this paper distinguishes between two classes of data used by a subsystem, confidential and nonconfidential. Confidential data cannot be retained in any private subsystem segment, nor may it be copied by the subsystem into any global segment, unless declassified by the customer. One of our conclusions is that a confined subsystem of the type described here cannot be permitted to output any data, even that tagged nonconfidential, unless it can be certified as error-free. In our present research, we are examining possible programming restrictions according to which nonleakage of confidential data can be guaranteed without the requirement of a program correctness proof.

It is interesting to note that Fenton has recently reported on a closely related problem, memoryless subsystems [8]. To provide a context within which he can prove rigorously his results, Fenton poses the problem on an abstract automaton (a Minsky machine). In this context, he showed how to guarantee confinement of confidential data when the machine's registers have been partitioned permanently into two sets -- those for confidential data, and those for nonconfidential. His implementation suitably restricts the programs for the machine to deal with the confidential Boolean problem. His proof demonstrates the impossibility of copying information from the confidential to the nonconfidential registers. Fenton also considers variable confidentiality classes. He shows that if there exists a register whose confidentiality can be changed (viz., from nonconfidential to confidential), it is possible to construct a program which will be able to leak private information. This latter result is similar in nature to ours, in that proof of nonleakage for variable confidentiality class machines is tantamount to a program correctness proof.

Acknowledgement. We are grateful to R. Stockton Gaines and H. D. Schwetman for helpful insights while preparing this work.

NE RIEN INSCRIRE :

Première ligne de texte

References

1. Lampson, B. "A Note on the Confinement Problem," Comm. ACM, 16, 10, Oct. 1973.

2. Graham, G.S. and Denning, P.J. "Protection - Principles and Practice," AFIPS Conf. Proc., 40, 1972 SJCC.

3. Feustel, E.A. "On the Advantages of Tagged Architecture," IEEE Transactions on Computers, C-22, 7, July 1973.

4. Weissman, C. "Security Controls in the ADEPT-50 Time-Sharing System," AFIPS Conf. Proc. 35, 1969 FJCC.

5. Fabry, R.S. "The Case for Capability Based Computers," Fourth Symposium on Operating Systems Principles, Oct. 1973.

6. Conway, R.W., Maxwell, W.L., and Morgan, H.L. "Implementation of Security Structures in Information Systems", Comm. ACM, 15, 4, April 1973.

7. Morris, J.H. "Protection in Programming Languages," Comm. ACM, 16, 1, Jan. 1973.

8. Fenton, J. S. "Memoryless Subsystems." Computer J. 17, 2 (1974), 143-147.

NE RIEN INSCRIRE :

Début du texte pour la 1ère page seulement

Dernière ligne de texte

NE RIEN INSCRIRE :