ON WORST-CASE COSTS FOR DYNAMIC DATA ELEMENT SECURITY DECISIONS

by

Franklin G. Woodward and Lance J. Hoffman

Memorandum No. ERL-M413

26 September 1973

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ON WORST-CASE COSTS FOR DYNAMIC DATA ELEMENT SECURITY DECISIONS

Franklin G. Woodward

and

Lance J. Hoffman

Department of Electrical Engineering and Computer Sciences
Computer Science Division
and the Electronics Research Laboratory,
University of California, Berkeley, California 94720

Research sponsored by the National Science Foundation, Grant GJ-36475.

Introduction and General Concepts

A major problem faced by users of most computer systems today is the lack of adequate file security. Early systems provided virtually no protection of stored data and it was often a quite simple matter for a user to obtain stored information belonging to another user without obtaining any particular permissions for such access [1, 2]. In more recent systems, some provisions have usually been made to distinguish which users have access to particular files, and what types of operations they may then perform. Access has been conditional upon the requestor's identity. If permission for further processing is granted, the appropriate type of access to the entire file is granted; if permission is denied, no file processing is allowed to occur.

Conway, Maxwell, and Morgan [3] have shown that this approach is frequently inadequate. There are many instances in which a file may contain data of varying sensitivity, some of which must not be divulged except under explicit conditions dependent on the data itself. As an example, consider an employee file. Such a file may contain a distinct record for each employee, all data pertinent to the employee being located in that record. This data may consist of the employee's name, social security number, salary, medical history, and any number of other fields. This file contains information relevant to a number of different users within the organization, but at the same time some information is present to which most of these users should not be allowed access. A few examples should clarify this point:

1) An employee may be allowed to access his own record, but no others.

2) The company doctor may be allowed to see all fields of all records excepting the salary field.

3) A clerk may require access to all fields within a record, but management wishes to restirct the clerk's inquiries to employees earning less than $15,000.

Of course the number of different instances is potentially very large, but at least two points are clear:

1) The access decision, when considered at the file level, is an all or nothing situation. A user may access either all data in the file or none. This is frequently inadequate.

2) It may often be desirable to control allowed data access at the sub-file level dependent on the specific authority of the particular user, and the particular value of the datum.

Selective security requires the file system to maintain some information on who gets access to what and how. This can be accomplished by use of a form of security matrix in which the rows represent users or classes of users and the columns specify the datum to be controlled. Each matrix element would then represent a decision rule to be enforced when user X requests access to datum Y (see Table 1). This scheme may be viewed as a subset of Lampson's object system [4]. If the decision

USER	EMPLOYEE	JOB NO.	SALARY	MED. HISTORY
A	R	R, W	-	-
B	R, W	R, W	R, W	R, W
C	-	R	-	R

Table 1. Sample data access control matrix for employee file. R = READ; W = WRITE (adapted from Conway et al)

rules are simple yes/no indications, they may be represented as bits and save memory space. This is not to underestimate the problems of matrix representation; on the contrary, the size of such a matrix is not a trivial concern and must be considered carefully.

The implementation of selective security can be effectively handled if capability for an access decision is provided at the sub-file level. Hsiao's system [5] controls access at the record level by use of "authority items." While this provides control over narrower domains than files, it still does not permit an access to be made dependent on the value of the data itself. Hoffman [6] used procedure-based formularies to gain this selective control. While this may provide the requisite protection, it creates still another problem. Specifically, if we must always make this decision whenever accessing any data element, an overhead burden is created that may not be tolerable in many applications.

To circumvent these limitations, Conway et al have proposed that a distinction be made between those access decisions which are independent of data values and those decisions which can only be made based upon the particular data value being accessed. The first type of decision may be considered data independent and the second data dependent, each for obvious reasons. These distinctions are then contained in the security matrix for both fetching and storing data.

The data independent decision need only be invoked once per file to ascertain if the user is allowed access to the requested datum. There is no need to repeat the request at each access and costly execution time decisions may be avoided. Obviously the data dependent checks may only be made at run time when the datum is available for scrutiny. Because such a check must be made for each access, the overhead is significantly higher than for the data independent check. The wise user will therefore use data dependent restrictions only when the less costly data independent checks are inadequate.

Conway, et.al. implement file security via five basic modules:

1. security matrix - contains the access rules
2. F_t - function to check data independent fetch requests at translation time
3. S_t - function to check data independent store requests at translation time
4. F_r - function invoked by data dependent fetch request at run time to check current data values
5. S_r - function invoked by data dependent store requests at run time to check current data values

These modules are used by the translator in the following manner: Whenever a source code reference is made to a previously unchecked datum, a call is made to F_t or S_t. F_t or S_t then consults the security matrix and determines if a data independent check is to be made. If so, and the fetch or store is permitted, then the normal object code is generated. If the fetch or store is not permitted under any conditions, then the translation is aborted and an improper access has been thwarted. If a data dependent check is to be made, then a call to F_r or S_r is inserted into the object code of the program and the check is postponed until run time.

During execution, items which have been checked by F_t or S_t and found to be data independent have no further restrictions placed upon them and will suffer no performance degradation. Data dependent elements, however, will invoke the special F_r or S_r routines. If access is permitted, the value of the datum is returned to the program. If not, a null of some sort is returned.

A few additional comments are called for at this point. First, this model assumes that the operating system under which it is running is secure with no bugs or hardware failures. Another assumption is that the user's identification has been authenticated elsewhere, although such authentication could perhaps be incorporated within the model. In a security system based upon user privileges, this is certainly an important matter to be considered. Also, there must be provision to protect the security matrix. This could be done by making the matrix an object within the security matrix itself, and allowing only restricted access. Finally, Conway et al suggest that encryption/decryption modules are easily implemented in this model, being easily incorporated into the F_r and S_r functions.

While the concepts embodied in this model are important and quite useful, the originators' implementation suffers from the following shortcomings:

1. All requests for access to a data base must be entered as source input to the system at translation time. While this may be enforceable, it may often be quite inconvenient. There is also an additional burden: one must somehow guarantee that the user executing the object code is the same user that initiated the source code translation.
2. The translated form of the requests must be held in a secure

manner so that they may not be altered by the user after the translation time checks. While this may be possible, increased vulnerability to tampering cannot be denied.

3. The access control rules must not change between translation time and any execution of the object code. Most changes would require costly retranslation of the source code to guarantee protection, even for normal production programs.
4. Calls to the data dependent routines must be generated in either of two ways. The first is by modifying the source translator - a highly non-trivial task in current compilers. The second is by preprocessing the source deck, which adds yet another step to program preparation and creates yet another level of code to be secured.

Conway et al recognized these problems and briefly discuss an alternative. They suggest that the F_r/S_r security modules could be implemented by adding a security matrix routine and modifying the I/O service routines of the operating system to perform the F_r/S_r functions. This would make the security independent of the translator employed but would necessarily require all data accesses to be checked at run time. In other words, F_t and S_t would no longer exist and the same execution time degradation problem which plagues Hoffman's formularies [6] would be present here.

An Implementation

The concepts of Conway, et al's model are quite appealing. There has been very little work done, however, to substantiate or refute the viability of such an approach. We therefore decided to test the various concepts by altering an existing operating system, the 6400 CALIDOSCOPE system (a variant of SCOPE) on the CDC 6400 at the University of California, Berkeley.

Our general approach was to utilize the basic concepts of Conway et al's model, but to eliminate its most restrictive assumption - that all requests for access to the data base be entered into the system at translation time. This goal was realized by having the security subroutines determine at execution time if data dependent or independent checks must be made and then proceed accordingly.

Most widely used general purpose compilers, on encountering an I/O request in the user program, generate a call in the compiled code to a routine of the system subroutine library. It is the system routine that performs the required I/O operation. The system routine can be programmed to make calls to special security subroutines. It is not necessary, therefore, to modify the compiler, but only the system I/O routines called. This has 3 major advantages:

1. Simplicity:

   The compiler is extremely large and complex. Any change might present formidable debugging problems. The system I/O routines are considerably easier to alter.

2. Reduced expense:

   A complete assembly of a FORTRAN compiler takes approximately 7 1/2 minutes of CDC 6400 time. At $7 per minute, this assembly costs about $50. On the other hand, the cost of assembling a system routine is usually under $10.

3. Generality:

   Modifying the system routines is actually more general, "cleaner", and provides a degree of compiler independence.

This approach was described by Conway et al as an alternative to their model for essentially the same reasons. However, they intended this avenue to be used only for data dependent checking (not data independent), and probably minimized discussion of it on this account.

The operation of this scheme proceeds conceptually as follows (see Figure 1). The system I/O routine maintains a pair of status bits for each file the user wishes to access. The first bit is a "first call" bit indicating if the file has been previously accessed during this run; thus the system I/O routines can make a data independent check on the initial request for file access. As in the Conway et al model, the data independent check may continue or abort the job, or determine if data dependent checks are to be made at each access. If data dependent checks are required, the second status bit is set to indicate calls to F_r and S_r are to be made. At each call to a system I/O routine from the user program, the routine checks the "first call" bit and then either calls F_t/S_t or proceeds with the next check. This check is of the second status bit to determine if a data dependent test is necessary and the routine either continues normal processing or calls the F_r/S_r function.

Such a scheme does increase the I/O time for processing a data independent file over the I/O time of a normal, unprotected file. Whether the gain of this security capability outweighs the added execution time penalty is for each installation to determine. The added execution time for data dependent checking is even more significant, but selective security is provided for those applications that can afford the cost. Finally, for this scheme to be effective the users must not be capable of creating their own I/O routines. A user with such a capability would be able to circumvent all of the proposed security checks.

This approach was taken and an actual implementation programmed for the CDC 6400 at Berkeley, principally because of availability and familiarity. This choice is not too restrictive as the same I/O concepts are employed on many computers. The particular system routines secured were those for formatted sequential input/output of the FORTRAN "RUN" compiler, specifically INPUTC and OUTPTC. This subset of I/O routines are estimated by the computation center staff to be invoked by approximately 75% of the compilations submitted at the campus computation center. For our tests, other I/O routines (binary I/O for example) were blocked by subprograms written to masquerade as those system routines. When called, these subprograms were set up to abort processing with the appropriate error messages.

The system I/O routines are quite complex assembly language subprograms that are entered multiple times for each I/O operation. Modification by a person unfamiliar with the routines is non-trivial. While a "working solution" (as opposed to a research project) would demand this effort, this project did not require such rigor. Therefore a still simpler implementation scheme was used: intercepting calls to the system I/O routines, performing whatever checks are desired, and then allowing I/O to proceed (see Figure 2). This scheme had the advantage of eliminating the need to modify any system routines and required coding only new and completely independent modules. Isolating the actual target system routines in absolute form on a private file had an additional advantage of making the project immune to any on-going systems library development.

The cost of this "linkage" type scheme over direct systems routine modification was, we hope, only a slight lessening of execution efficiency.

Security Matrix Content

The security matrix was maintained as a data file in secondary storage. The matrix was accessed only by the data independent security module. Reference to the matrix was made on the first I/O request to a file by the user. Actual storage and retrieval of the matrix was done in a fairly simple-minded manner, with only minor concern for matrix protection. This is, of course, totally inadequate for a real world situation, but it served the purposes of this cost study well enough. Because the contents of the security matrix reveal most features of the implementation, a close look at the matrix elements will be beneficial here (see Figure 3).

The matrix is actually a sequential file with separate records for each user of each protected file. The file protection system provided using this matrix may best be typified as essentially open [7], with an authority or access control list structure [8]. The absence of any records concerning a given file implies that the file is not protected and cues the system to do no further checking of I/O for this file. If a file is listed in the matrix, but there exist no records for the user seeking access, it is assumed that user has no privilege to the file and cues the abortion of the current process. If, on the other hand, a record is found in the matrix for the particular file and user, the user's access is constrained by the information present in the matrix record (see Figure 4).

Besides the file and user names, the matrix record contains the following data. One character indicates data independency for read access. If it is a one, then this user is granted unrestricted read access to the file for the remainder of the run, and no further information from the security matrix is required. If the indicator is zero, then another character is checked to see if data dependent read access is allowed. If this character is also zero, then no reads of this file are permitted by this user (any attempt to read will terminate the users process). If the data dependent character is non-zero, however, its value indicates how many fields must be checked at each read access (the maximum is arbitrarily set at 6 fields). An analagous pair of characters indicate write access capability.

The matrix also contains the record length of the subject file. It is used to set up a "buffer" visible to the data dependent checking module when system I/O transmits file data to the user program. The record length in the matrix overrides whatever record length the user has indicated in his FORMAT statement.

A failure mode indicator is also present which specifies what to do upon the failure of a data dependent check. If its value is one, then BCD zeroes are written into the appropriate field in the "buffer". If the indicator is zero, then the entire "buffer" is zeroed. The implications of this indicator are quite significant. It not only allows withholding of particular data elements from a user, but also may deny access to the entire record if the test of one field fails. An employee file again serves as a useful example. If a user is allowed access to records dependent upon the contents of the salary field, this option may have the following effects. If the user's allowed access range is null and zeroes are always returned for salary, then allowing access to the rest of the record is harmless and perhaps worthwhile. However, if the user may access records only when salary is less than $15,000, then a zero value returned indicates that the subject makes $15,000 or more. If the entire record is not zeroed, then the subject's identification is known and a fact concerning a particular individual's salary is revealed. Zeroing the entire record would help conceal the subject's identity. Our implementation used this option for both read and write access control to measure comparitive costs. Real world systems, however, should not allow any output to occur after a write request has been denied.

To handle data dependent checks, the security matrix must also specify characteristics of the fields to be checked. A pair of matrix elements thus specify the starting and ending character positions of each data dependent field in a record. To provide a very simple variable range check another pair of elements gives the minimum and maximum values of the permitted access ranges for each field. It should be noted here that this implementation only allows data dependent checking on integer fields. While restrictive, it could be fairly easily extended to alphameric data without unduly increasing execution speed. This extension was not done due only to a lack of time.

The security matrix is really the heart of the security system. By including more or less information in it, the complexity of the potential data dependent checks can be controlled. The current implementation has included those elements thought to be useful in a general sense and which offer a fair range of control. What has been included here should not be considered definitive. Furthermore, one should bear in mind the relationships between record length, quantity of records, and file length. Because large sequential files portend long access times, it may be necessary to consider faster storage formats for the security matrix.

Another problem which had to be dealt with was user and file identification. The methods of guaranteeing identification are quite relevant in discussing a security model, but are here assumed to have been provided for. The user must be identifiable in order to determine his access rights. Such identification can normally be given in the user's job control statements (JCL) , along with specifications of those files which the user intends to access. But a second check is also necessary to guarantee that the file actually addressed is the same as the user specified. This is normally automatically done using file labels (again a secure procedure is assumed). Label processing on the Berkeley 6400 is (can you believe it?) only in the development stages. As there is no reliable means of obtaining file identifiers from the system, simulation of labels and JCL was done in our experiments.

A summary of the basic logic necessary to perform the security checks on system input is shown in Figure 5. System output uses an analagous flow chart. This logic is invoked by the linkage created between the compiled user program and the real system I/O routine. A variation from the Cornell model which evolved during programming should be mentioned. Conway et al specifically designated dual translation time (F_t,S_t) and run time (F_r,S_r) modules as necessary to accomodate both data fetches and stores. The security modules of this implementation, however, are only enforcers of the decision rules contained in the security matrix. The data dependent check must only examine the contents of the data "buffer". The only difference between input and output is the timing of the check. Therefore only one data dependent module was written, whereas the Cornell model has two (F_r,S_r). Similarly, only minor differences occurred in the data independent checks, and again one module sufficed.

Determination of Costs

While there have been some recent efforts to establish generalized security-cost relationship models [9], little has been written of the actual execution costs of various security systems [10] or models. Cost determination has been a primary goal of this implementation.

The time to perform a FORTRAN I/O operation depends on several parameters, including record length, format, and argument list. All tests were therefore performed using identical I/O parameters. An interesting aspect of formatted FORTRAN I/O in the CDC RUN compiler is also relevant. Whenever a single I/O operation is to be performed, control is passed back and forth between the users object code and the system I/O routine a variable number of times depending on the length of the argument list. Specifically, there are n+2 transfers to system I/O, where n is the number of items in the argument list. This is important because the security linkage created only executes special "security code" on the first call of a READ and the first and last call of a WRITE. Obviously, the longer the argument list, the less will be the relative percentage of time taken in executing the "security code". We wished to obtain worst case results, therefore, only two arguments were specified for each I/O operation.

Another important facet of execution time concerns the efficiency of the coded security programs. In this implementation, only limited effort was spent in optimizing the source code. Because the linkages between the user programs and the system I/O routines are entered so frequently, they were coded in CDC's assembler language, COMPASS. The data independent and dependent modules, however, were coded in FORTRAN and may contain extraneous code. In any event, diligent code optimization undoubtedly will lower the tested execution times.

Our actual cost experiments fell into three categories:

1. A set of timing tests to determine the cost of the initial file I/O request.
2. A set of timing tests to determine the cost of each subsequent file I/O request.
3. An analysis of the main memory required for the security subroutines and tables.

Cost Component I: Initial File I/O Operation

The first I/O operation initiates the call to the data independent security module, references the security matrix, and initializes several variables. Table 2 shows the basic times required to execute code to determine data dependence/independence, initialize variables and perform one read or write. The times of Table 2 are quite trivial (approximately 12 to 26 milliseconds). The time required to search the security matrix is not included as it is variable depending on the position of the subject file in the matrix.

EXPERIMENT NUMBER	TEST OF	NUMBER OF FIELDS CHECKED	NUMBER OF CHARACTERS PER FIELD	NOTE	CPU BASE MILLISECONDS* READ	CPU BASE MILLISECONDS* WRITE
1	DATA INDEPENDENT	-	-		13.72	13.72
2	DATA DEPENDENT	1 (FAILS)	10	1	13.16	12.28
3	DATA DEPENDENT	6 (ALL FAIL)	10	1	25.34	25.58

TABLE 2. Average time to perform initial I/O operation of file, not including time to locate file and user in security matrix

Note 1: All fields tested fail; failure causes only failing field to be zeroed.

* $\pm$ .02 milliseconds

Recall that the security matrix is currently stored as a sequential file in secondary storage. The variable search time was clocked to be 3.98 milliseconds per record in the security matrix. This is the time required for one iteration of a program loop, including a READ, to search file. The expected time to locate a particular record in the matrix is then:

E(search time) = 1/2 (number of records in matrix)(3.98 milliseconds)

This time should be added to the times given in Table 2 to obtain the total expected time to perform the initial I/O operation of a file. Clearly the expected time for the first read is almost totally dependent on the size and organization of the security matrix. Because this operation is only performed once per file per run, its time may not be significant in the overall picture. Furthermore, improved organization for matrix storage might dramatically reduce the average search time.

Cost Component 2: Subsequent File I/O Operations

This component was determined by measuring the time taken to do the last 50 I/O operations out of 51 performed, and then computing the mean. The first I/O operation is not included. Table 3 shows the mean times to complete an I/O instruction under a number of different conditions.

Experiment 1 gives benchmark CPU times to do read and write operations in the standard unprotected environment.

Experiment 2 indicates the mean I/O time when the subject file allows data independent access. The increase is significant (READ 22%, WRITE 32%). It is felt, however, that a large portion of this time may be due to the linkage method of implementation. Had the linkage modules not been created and the decision code placed directly into the system I/O routines, a significant amount of source code could be eliminated. The effective increase in time, therefore, would probably be much less in a real world implementation. It certainly calls for further study, and points out that we really are describing here a worst-case situation.

Experiments 3-8 show the I/O time incurred when various data dependent checks are made. As can be expected, the time increases fairly linearly with the number of fields to be checked. The effect of a data dependent check failing is also evident as a slight time increment to zero the field. What is interesting is that the incremental time necessary to perform a one-field data dependent check is not very large. In a real world implementation, this time would be further reduced by the same amount as would the data independent check (see above). Selective security on a limited basis might prove to be quite reasonably priced.

EXPERIMENT NUMBER	TEST OF	NUMBER OF FIELDS CHECKED	NUMBER OF CHARACTERS PER FIELD	NOTE	CPU MILLISECONDS PER I/O* READ	CPU MILLISECONDS PER I/O* WRITE	% INCREASE OVER BENCHMARK READ	% INCREASE OVER BENCHMARK WRITE
1	BENCHMARK	-	-		3.22	2.40	-	-
2	DATA INDEPENDENT	-	-		3.92	3.16	22	32
3	DATA DEPENDENT	1 (PASSES)	10		4.24	3.70	32	54
4	DATA DEPENDENT	2 (ALL PASS)	10		4.72	4.24	47	77
5	DATA DEPENDENT	6 (ALL PASS)	10		6.74	6.24	109	160
6	DATA DEPENDENT	1 (FAILS)	10	1	4.38	3.76	36	57
7	DATA DEPENDENT	2 (ALL FAIL)	10	1	5.08	4.40	58	83
8	DATA DEPENDENT	6 (ALL FAIL)	10	1	7.72	6.82	140	184
9	DATA DEPENDENT	1 (FAILS)	10	2	5.34	4.40	66	83
10	DATA DEPENDENT	6 (LAST FAILS)	10	2	7.92	6.90	146	187
11	DATA DEPENDENT	1 (FAILS)	5	1	-	3.56	-	48

TABLE 3. Average times per I/O operation (not including the first I/O operation).

Note: 1 All fields tested fail; failure causes only failing field to be zeroed.

2 Last field tested fails, others pass; failure causes entire record to be zeroed.

* $\pm$ .02 milliseconds

Experiments 9-10 highlight the sensitivity of a failing data dependent field to the type of zeroing specified. Whereas failure in experiments 6-8 caused only the failing field to be zeroed, a failure in experiments 9-10 caused the entire 70 character record to be zeroed. As can be expected, the times are significantly increased (up to one millisecond) if the entire record is zeroed.

The final experiment of Table 3 showes the effect of reducing the field size to be examined from 10 characters to 5. Because the "buffer" used for data checking is organized on a one character per word basis, it is predictable that reducing the field size should reduce the time required to perform the check. The time of this check, and failure, is now only about 13% greater than for the data independent write.

Cost Component 3: Memory Requirements

The final relevent statistic is the memory space required for the added routines. The test runs of this implementation required 15622_8 words of extra memory beyond the normal memory requirement. This is quite large, but is a function of the particular implementation. The space required to store the absolute library of system routines was 5113_8 words, and would be reduced in a real application due to the currently redundant subprograms. Also the method used to store the file information taken from the security matrix is extremely wastefull of space and occupies 5361_8 words. This is because it is an array dimensioned to handle 100 files and stores all data elements as whole words rather than bits. Rather simple methods could reduce the required space by 4000_8-5000_8 words. Finally, the linkages themselves take 1643_8 words. Much of the code is redundant of the actual system I/O routines, and would be eliminated when the linkage decisions were actually placed into the system routines. All matters considered, the total memory requirement would be considerably reduced in the course of incorporating this implementation into the system functions (see Table 4).

	OCTAL MEMORY WORDS	
	CURRENT "WORST CASE"	EXPECTED "REAL WORLD"
Library of system routines	5113	4200
Array of user file restrictions	5361	1000
Security modules	1263	1263
Linkage modules	1643	300
TOTAL	15622	6763

TABLE 4. Comparison of the current implementation's octal memory requirements with the expected requirements of a "real world" implementation.

Preliminary Conclusions

The test results indicate that modification of system I/O routines may be an effective method of providing selective security at reasonable cost. The difference between benchmark and "worst case" data independent I/O service times is encouraging. If this difference can be made negligible, then the viability of performing data independent checks at run time would be shown. As indicated, a more thorough implementation might quite conceivably reach this goal.

Nevertheless, caution must be used in viewing the test results. The operating system under which this test ran is tailored for a specific environment. Proof of applicability to a wider class of machines would be desirable. Also the I/O routines upon which this implementation was built may not be typical of those used for heavy I/O processing. Formatted FORTRAN requires rather specialized I/O and does not provide high data rates. Again, it would be desirable to demonstrate applicability of this implementation to a wider class of languages.

The authors are currently investigating a more refined implementation of our selective security procedures on an IBM 360/67. Initial study indicates high feasibility. Emphasis will again be placed on cost determination and evaluation.

Summary

Conway, Maxwell, and Morgan have pointed out the need for allowing differing access abilities at the data element level to different users. Previous efforts to provide such security have suffered a heavy overhead burden due to the need to scrutinize all data elements at each run-time access.

In order to implement a model utilizing the properties of data dependency/independency, Conway et al described four functions and a security matrix. Each row represents a user, each column represents a data element, and each matrix cell indicates the access rights of a user to a data element. One pair of data fetch and store functions provide a translation time check for data independent conditions. A second pair of functions are invoked at each data access if the data access permission has been specified as data dependent.

We have discussed a number of problems with this approach, all relating to the translation time security procedure. Our implementation provides data dependent and independent checks, but both at run time. It is based upon modifying the operating system I/O routines to perform a data independent check on the initial requested access to a file by a user. This check interrogates a system security matrix and specifies one of three courses: 1) all future accesses of this file are allowed and there shall be no further access obstructions; 2) no accesses are allowed and the user job is terminated; 3) all accesses are dependent upon the value of the data being accessed and a check will be made upon each following access.

Our approach was then tested experimentally to determine average CPU overhead costs for different file restrictions. The results indicate that data independent accesses are about 22-32% less efficient than the corresponding accesses in an unprotected environment in the worst case. There are some indications that under improved conditions, the difference in performances might be made negligible. When the security matrix is stored as a sequential file the time to actually perform the data independent check varies almost directly with the size of the security matrix. Improved organization here might dramatically cut down on costs. Also, the time required to perform a data dependent check varies with the number of characters inspected. Simple data dependent checks may be made at a fairly low incremental cost.

The test results indicate that an effective and cost-feasible method of selective security may be possible in a real world operating system. Current investigation is being made to determine the viability of the given approach on computer systems other than the CDC 6400 and for high level languages other than FORTRAN.

Acknowledgments

We are indebted to R.S. Fabry for his helpful discussions and insights, and to T. Sumner of the Campus Computer Center at the University of California, Berkeley for his programming design suggestions.

References

1. Hoffman, L.J., Computers and Privacy: A Survey, Computing Surveys, 1, 2 (June 1969), 85-103.

2. Friedman, T., The Authorization Problem in Shared Files, IBM Systems Journal, 9, 4 (1970), 248-280.

3. Conway, R.W., Maxwell, W.L., and Morgan, H.L., On the Implementation of Security Measures in Information Systems, Comm. ACM 15, 4 (April 1972), 211-220.

4. Lampson, B.W., Protection, Proc. Fifth Annual Princeton Conference on Information Sciences and Systems, Princeton University, Princeton, New Jersey, March 1971, 437-443.

5. Hsiao, D.K., A File System for a Problem Solving Facility, Moore School Tech. Report #68-33, Univ. of Pennsylvania, May 1968, 175 pp.

6. Hoffman, Lance J., The formulary model for flexible privacy and access controls, AFIPS Conference Proceedings, 1971 FJCC, Vol. 38, 587-601.

7. Daley, R.C., and Neumann, P.G., A General-Purpose File System for Secondary Storage, AFIPS Conference Proceedings, 1965 FJCC, Vol. 27, 213-229.

8. Graham, G.S., and Denning, P., Protection-Principles and Practice, AFIPS Conference Proceedings, 1972 SJCC, Vol. 40, 417-429.

9. Turn, R., and Shapiro, N.Z., Privacy and Security in Databank Systems: Measures of Effectiveness, Costs, and Protector-Intruder Interactions, AFIPS Conference Proceedings, 1972 FJCC, Vol. 41, 435-444.

10. Friedman, T.D. and Hoffman, L.J., Encryption Time Requirements for Programmed Encryption Methods, Memorandum ERL-M378, Electronics Research Laboratory, University of California, Berkeley, 1 June 1973.

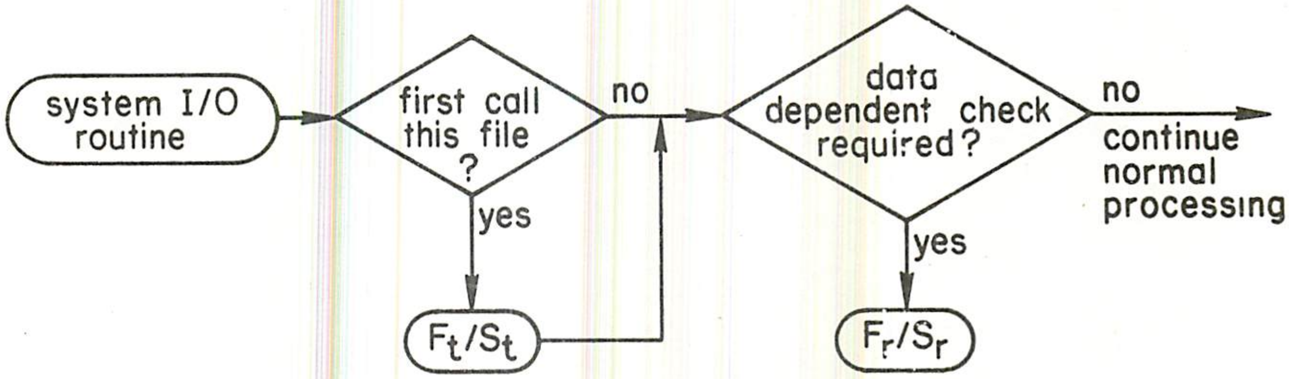


Fig.1 Basic logic for system I/O routine

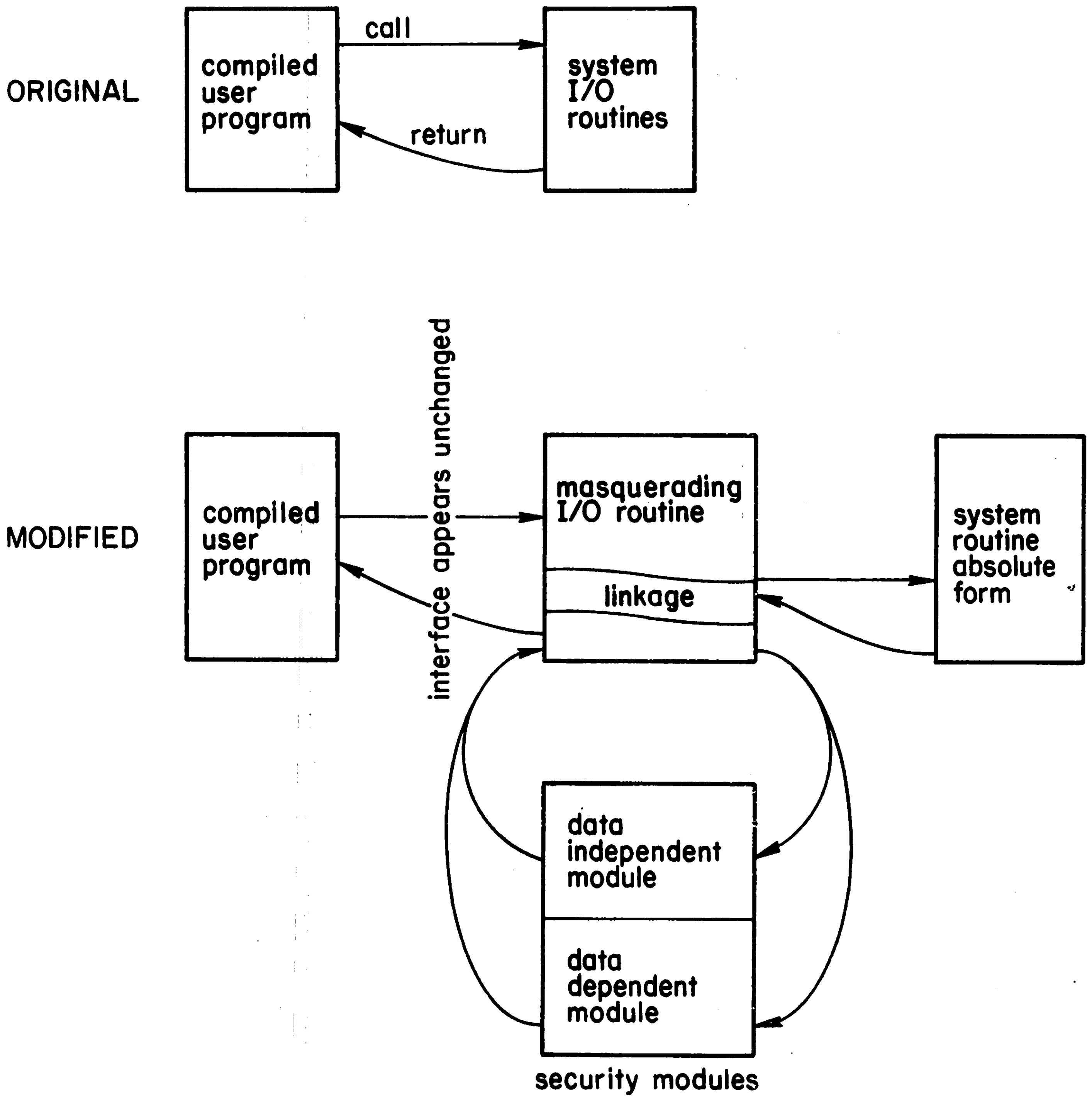


Fig. 2 Modified system I/O to implement data security modules

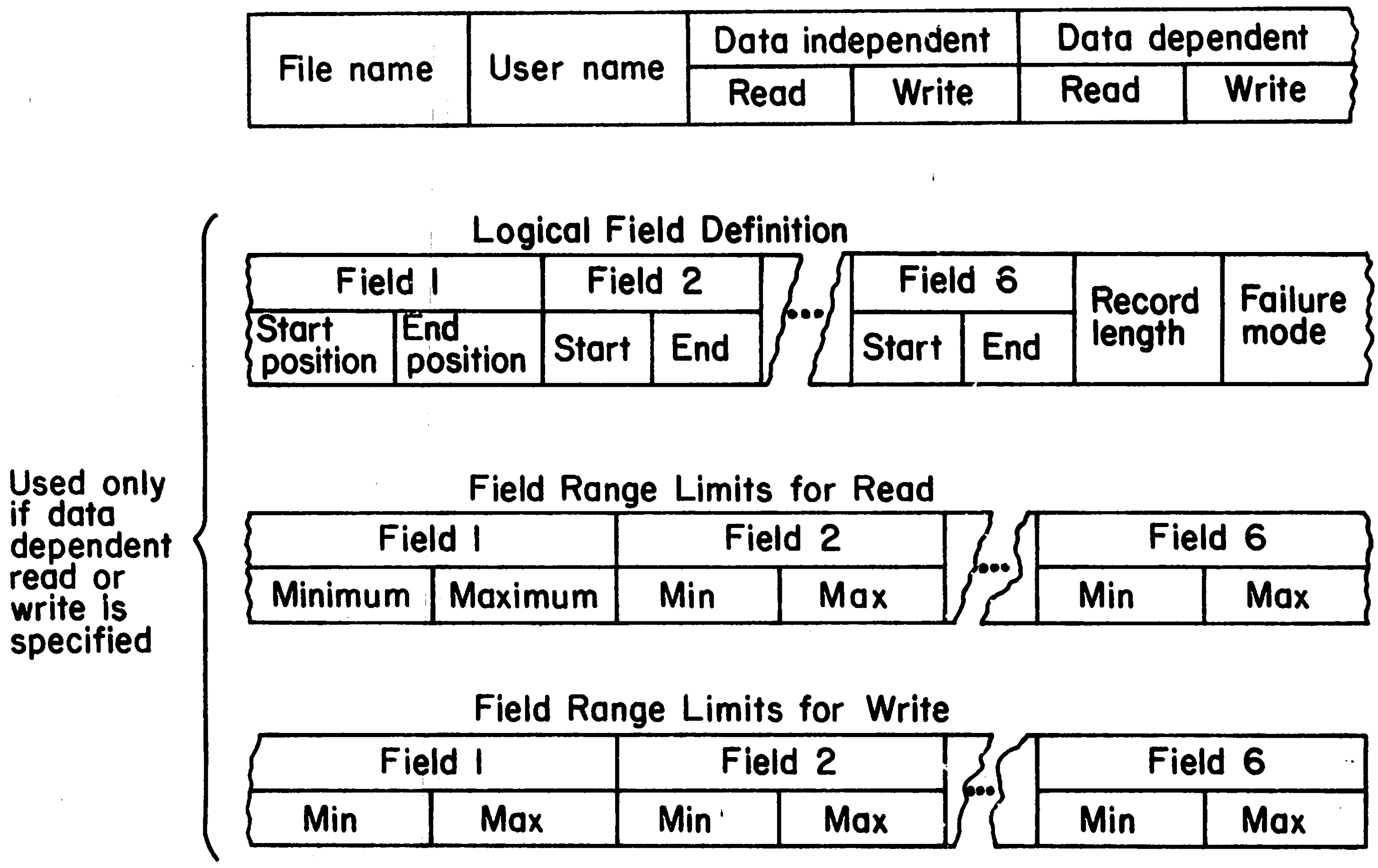


Fig. 3 Security matrix record representation

	File in matrix	File not in matrix
User in matrix	access subject to control	any access allowed
User not in matrix	access not allowed this user	
	File protected	File unprotected

Fig. 4 Access control using security matrix

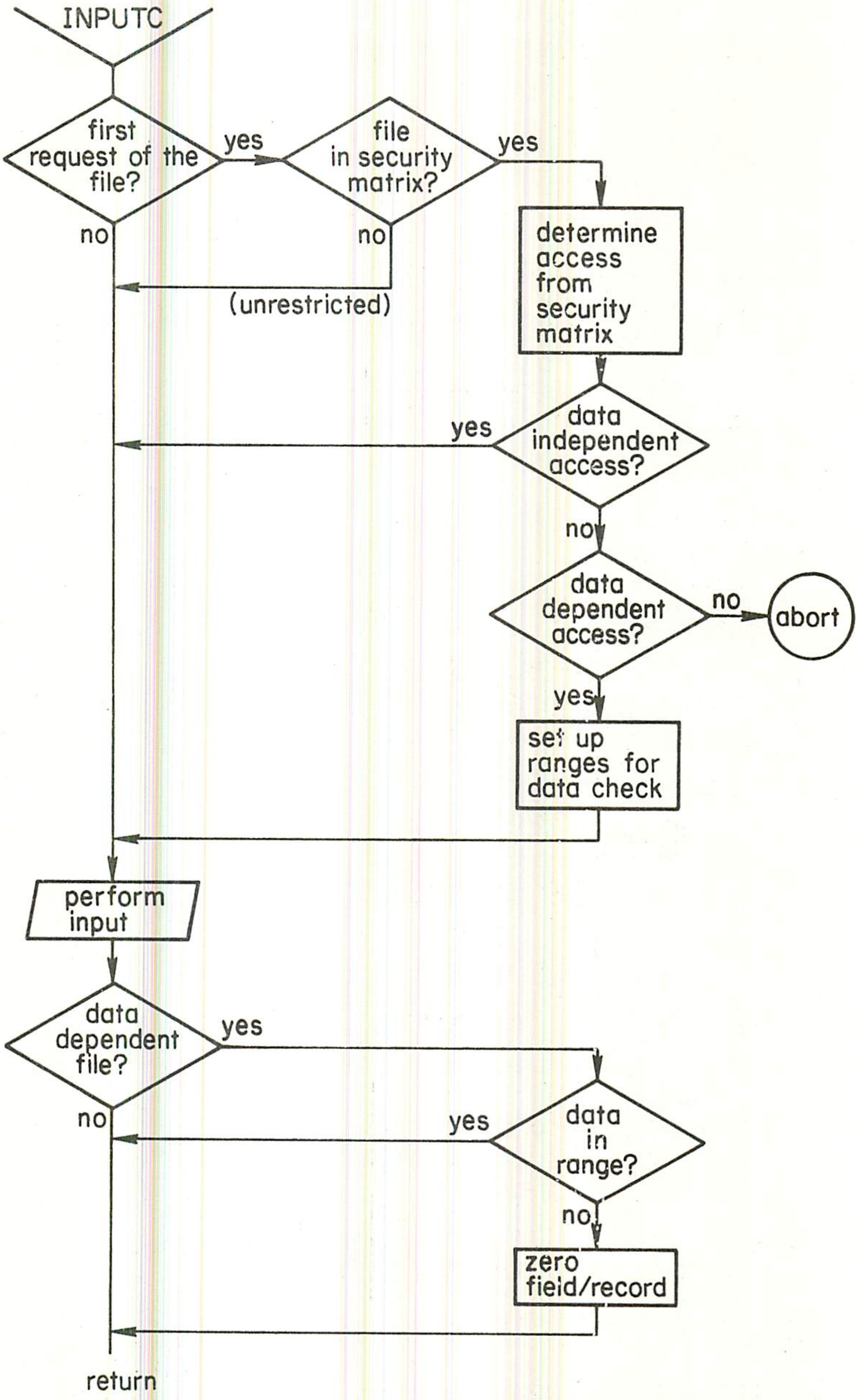
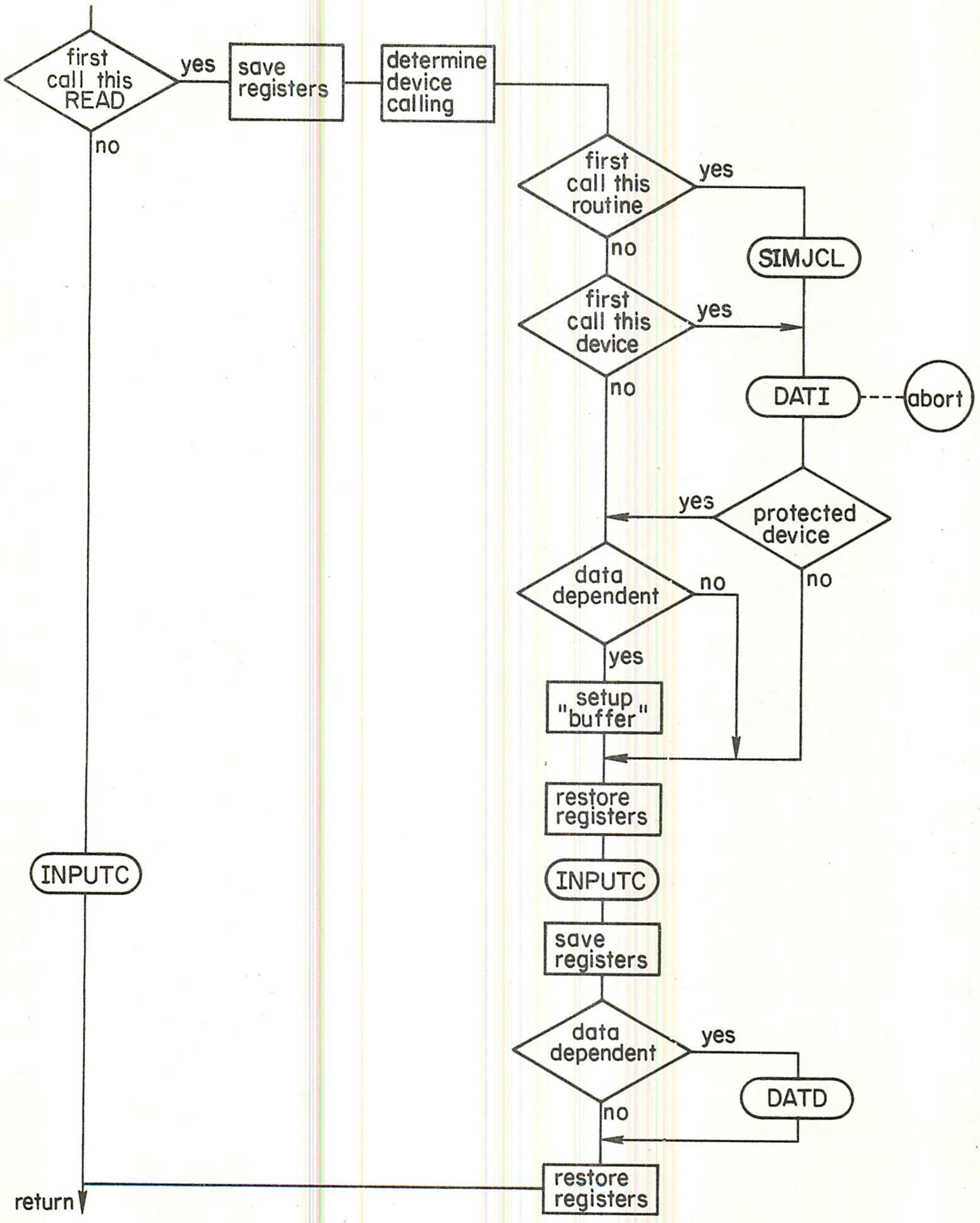


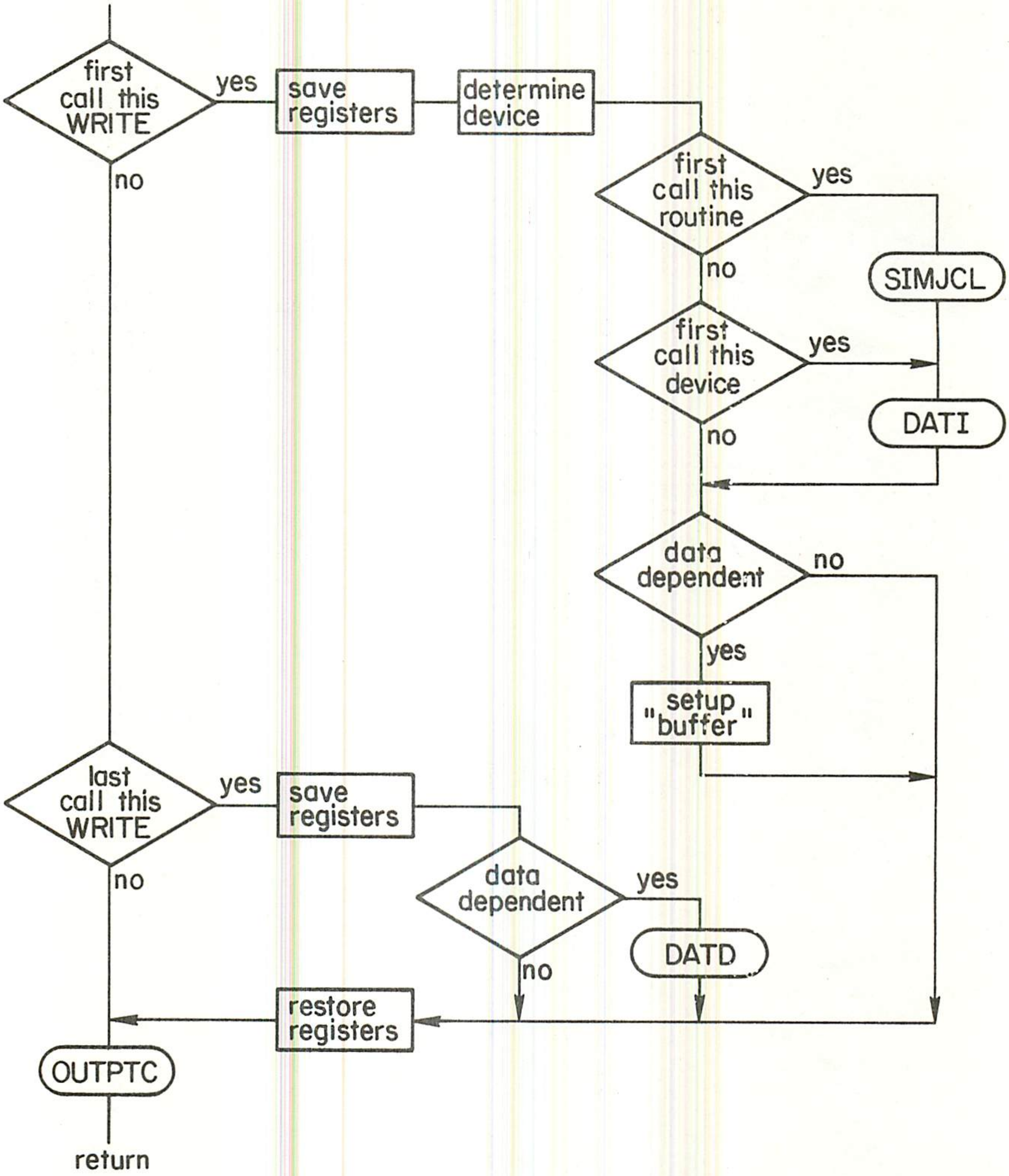
Fig. 5 Logic to perform security checks in system input

Appendix I

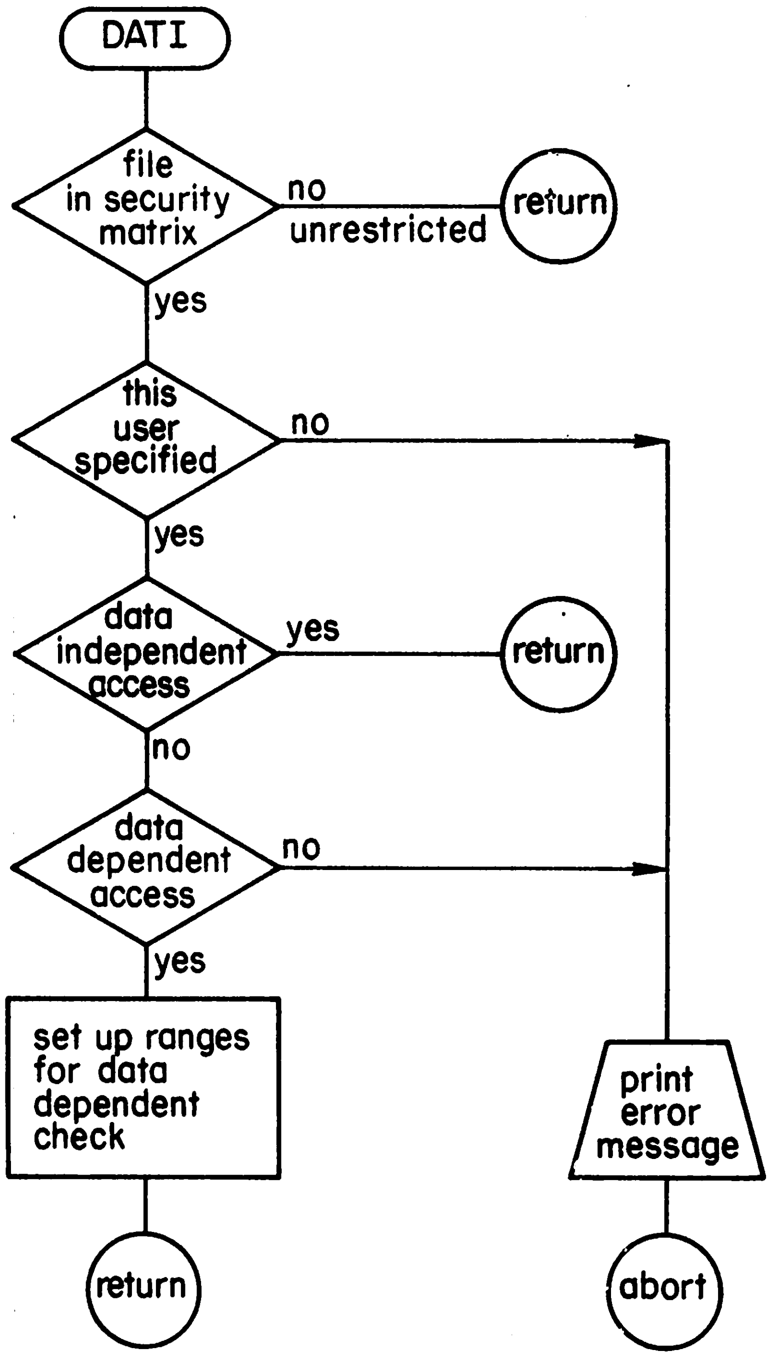
Flowcharts of Major Modules



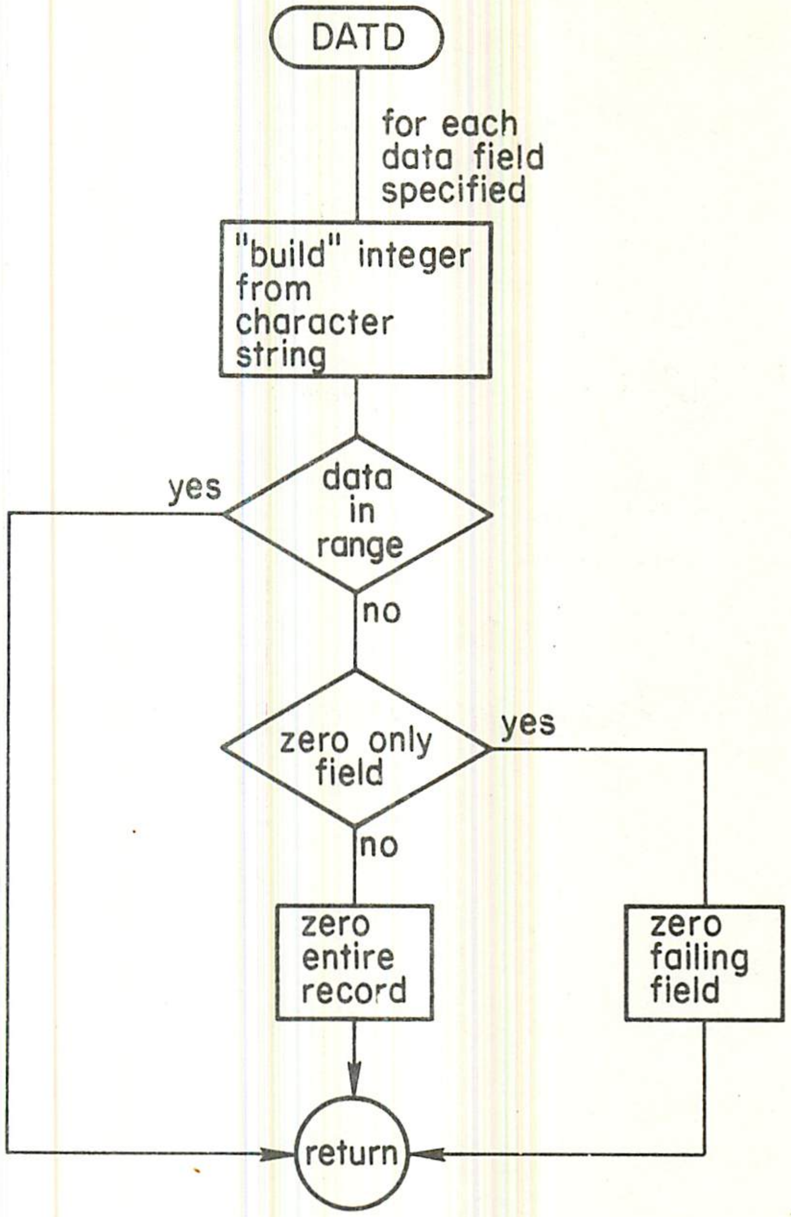
Logic of INPUTC linkage module



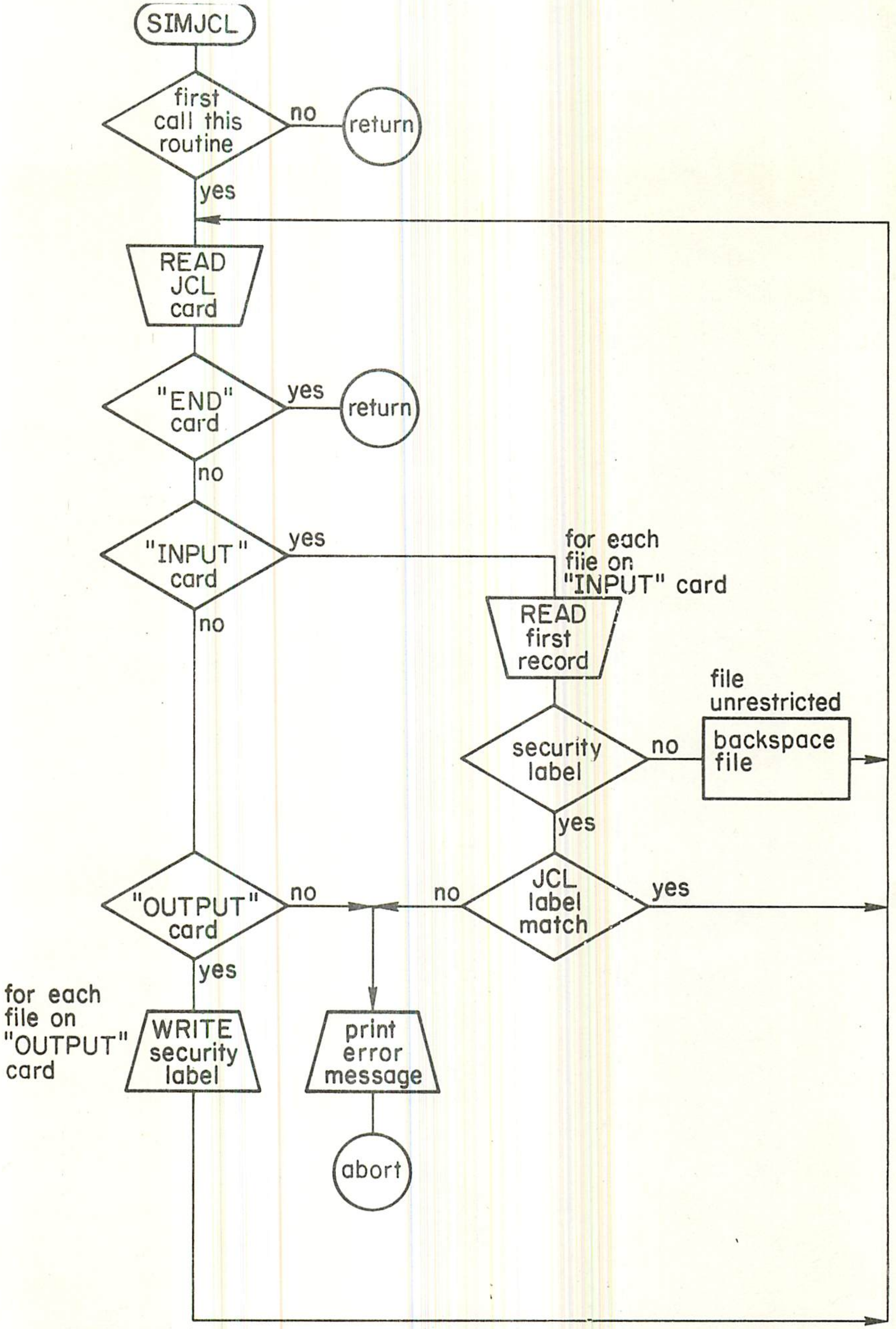
Logic of OUTPTC linkage module



Logic to perform data independent check (DATI)



Logic to perform data dependent check (DATD)



Logic to simulate JCL and label processing (SIMJCL)